UNIVERSITY OF LEEDS

Profiling Optimised Haskell

Causal Analysis and Implementation

Submitted in accordance with the requirements
for the degree of Doctor of Philosophy
to the University of Leeds, School of Computing

June 2014

Author:
Peter Moritz Wortmann

Supervisor:
Prof. David Duke

Advisors:
Dr. Simon Marlow
Dr. Satnam Singh
Prof. S. Peyton-Jones

The candidate confirms that the work submitted is his/her own, except where work which has formed part of jointly authored publications has been included. The contribution of the candidate and the other authors to this work has been explicitly indicated below. The candidate confirms that appropriate credit has been given within the thesis where reference has been made to the work of others.

This thesis is based on the same material as the paper "Causality of optimized Haskell: what is burning our cycles?", by the thesis author and David Duke, published in the proceedings of the 2013 ACM SIGPLAN symposium on Haskell. Relevant material is reproduced in Chapter 3, Chapter 4 and Chapter 5, and is outside of contributions in the form of supervision and discussion entirely the author's original work.

Some parts of Chapter 5 overlap with the master thesis "Stack traces in Haskell" by Arash Rouhani from the Chalmers University of Technology. This is due to our joint work on improving the stack tracing capabilities of Haskell, which is however not integral to the contributions of this thesis.

Acknowledgements

It goes without saying that this work would not have been possible without all the people around me putting rather unwise amounts of effort into encouraging me to do crazy things.

First off I have to thank David Duke and Microsoft Research for getting the ball rolling by arranging for pretty much everything I could have hoped for going into this PhD. Second I would especially like to thank my supervisor David Duke again for contributing quite a bit to making my academic life enjoyable, as well as playing an integral parts in the few productive bits of it. I always feel like he ends up reading my texts more often and more thoroughly than I ever could, which given his other commitments is an incredible feat.

Furthermore, I have to credit my advisers from (formerly) Microsoft Research for always trying to push me into the right direction. And even though we probably did not manage to advance much on the project's original goal, their feedback was always much appreciated. I have to especially thank Simon Marlow for arranging a number of meetings over the years that ended up shaping this work significantly.

Finally I have to thank all the busy people that found time to read through my thesis, and discovered just how many orthographic, grammatical and semantic errors a person can hide in a document of this size. In no particular order I would like to credit Richard Senington, Dragan Šunjka, José Calderón as well as my mother Dorothee Wortmann and brother Jonas Wortmann for their tireless efforts trying to digest my heavy prose.

Notes

For brevity, we will use the female form throughout this thesis whenever there are multiple conceivable ways of address. This is completely arbitrary (I flipped a coin) and in no way a reflection on the role of either gender in modern society.

No actual trees were harmed in the production of this PhD. Well, except for all that printing paper. My apologies.

Abstract

At the present time, performance optimisation of real-life Haskell programs is a bit of a “black art”. Programmers that can do so reliably are highly esteemed, doubly so if they manage to do it without sacrificing the character of the language by falling back to an “imperative style”. The reason is that while programming at a high-level does not need to result in slow performance, it must rely on a delicate mix of optimisations and transformations to work out just right. Predicting how all these cogs will turn is hard enough – but where something goes wrong, the various transformations will have mangled the program to the point where even finding the crucial locations in the code can become a game of cat-and-mouse.

In this work we will lift the veil on the performance of heavily transformed Haskell programs: Using a formal causality analysis we will track source code links from square one, and maintain the connection all the way to the final costs generated by the program. This will allow us to implement a profiling solution that can measure performance at high accuracy while explaining in detail how we got to the point in question. Furthermore, we will directly support the performance analysis process by developing an interactive profiling user interface that allows rapid theory forming and evaluation, as well as deep analysis where required.

Contents

Chapter 1

Introduction

"I would like a little more enthusiasm and a little less Latin!"

— *Asterix the Gaul, René Goscinny & Albert Uderzo*

Program performance is a rather uneasy topic for both application users as well as programmers. Most energy is generally expended on functionality: The more useful an application is, the more value it will have for the end user, and the more likely it is that the programmer will get paid for their work. However, the more features an application accumulates and the higher the complexity rises, the more likely it becomes that performance problems sneak their way into the program. At this point, things might slowly become more unpleasant – our user will grow impatient, while the software developers might be faced with the troublesome task of revisiting past design decisions in order to prevent eventual collapse. Every seasoned software developer knows from experience that without constant attention, basically any application under active development will slowly converge into a similar state.

As a result, program performance is rarely thought of as a positive program property. Instead, in software development we *fear* bad performance in the same way that we might fear an empty bank account: We know that no matter how much effort we put into saving up, we are never more than a few unwise decisions away from a hurtful reality check. And just like a good budget and responsible spending behaviour might give us the feeling of security, programmers also seek to use "safe" strategies in order to prevent the eventual performance catastrophe. Where doubts about performance creep in the shadows, we will try to stick to patterns that have served us well in the past. We will aim to avoid uncertainty, and stick to with what we perceive as "fast" software development practises. This makes performance a major hurdle for introducing new development methodologies. Raising doubts about performance viability can be as effective as suggesting that a political party might increase taxes: Even where the concrete effects are negligible, invoking the worst case can yield to dismissal by reflex.

Despite these problems, there has been a remarkable amount of interest in the ideas behind functional programming languages in recent times. And within this movement there is particular interest in writing fast real-world programs. A popular reason to cite is that we have found a new scarecrow: As has been known for quite some time, hardware is moving towards parallel architectures. And it is pretty clear that exactly the "safe" strategies for single-core performance will yield us programs that are exceptionally hard to parallelise. Functional languages promise a way out: A potential safe harbour where we can again find performance predictability.

1.1 Problem statement

However, at the most basic level this is still not solving the problem, but simply trying to dodge a bigger one. And without trying to oversimplify the matter – can we truly expect to be able to build fast parallel programs until we can assure predictable performance for sequential code fragments? For this thesis we will therefore attempt to take another stab at the classic problem of profiling sequential programs.

We will specifically target the programming language Haskell for this. As we alluded to in the introduction, the reason is that we see a problem with transparency: The development community seems split between a relatively small group of "crack" programmers squeezing impressive performance results out of the development infrastructure [Mainland et al., 2013] and beginner programmers struggling to solve even relatively basic problems without running into apparently inexplicable performance problems [Tibell, 2011].

Therefore our mantra for this work will be honesty: The Haskell infrastructure is of significant complexity, yet for performance purposes we can not simply treat it as a black box. We will aim to build a novel profiling infrastructure that attempts to convey the inner workings of a running Haskell program in all detail required to make sense of its performance characteristics. This especially means that we will take care to change program compilation and execution only as far as it is absolutely required for us to support performance analysis. Consequently we will opt for light-weight annotation strategies that are nevertheless robust against code optimisations. Furthermore, we will take care to use non-invasive instrumentation methods in order to directly measure program performance.

1.2 Structure

For this work we will develop a novel profiling solution for Haskell from the ground up. Our aim will be to have low overhead and stay optimisation-aware at the same time.

Implementing this will require quite a few subtle design choices, therefore in Chapter 2 we will fundamentally re-evaluate the question of what we should ask from a profiling tool in the first place. We will demonstrate that performance analysis is an instance of abductive cause/effect analysis, where our causes are program terms and effects are costs appearing at program execution.

For Chapter 3 we will then take a closer look at the Haskell development environment. We will flesh out the characteristics of the language and identify the core issue that we will be trying to address. After that point, we will start explaining the language's compilation and execution mechanisms in greater detail, focusing specifically on the Glasgow Haskell Compiler. This will lead into an overview of GHC's optimisation passes, as well as the development of an abstract performance model for Haskell programs.

At this point the stage will be set to get to the primary concerns of this thesis: Chapter 4 performs a formal causal analysis of the performance model developed in the last chapter. The idea will be that using counter-factual causality theory we can establish causality between the unoptimised program source code and the abstract cause terms of our performance model. As we will see, we can even make this work in the presence of program optimisations.

In Chapter 5 we will return into the real world to instantiate our abstract causality model with actual source code locations and costs. We will show how we can view different profiling approaches as reconstructing different subsets of the "full" cause terms. We will then go on to explain how we can implement our own profiling solution based on our findings. To this end we will not only have to modify most important transformation stages within GHC, but also settle for a suitable representation to allow our meta data to live alongside object code. Furthermore, we will explain how we can capture performance data using sampling, and assist the user in performance analysis using an extension to the ThreadScope profile analysis tool.

Chapter 6 will evaluate our approach, by measuring the different overheads involved in profiling, and contrasting it against the data collected. We will also consider an extended usage example for our profiling solution in order to convey an idea of how we imagine performance analysis with our tool would work.

Finally, Chapter 7 will summarise our contribution, review prior work and provide some remarks on possible future extensions.

Chapter 2

Background

“Well, it is an untidy sort of forest anyway. Trees all over the place!”

— *Asterix the Legionary, René Goscinny & Albert Uderzo*

Our idea of how fast programs look like has clearly changed. From the perspective of performance analysis and especially profiling tools, this means that we should also re-evaluate our role critically. Is putting time estimates across from function names truly the limit to what we can do? Maybe we could do a much better job if we took another hard look at what we are actually trying to accomplish? Clearly our job is to help the programmer in the development task – but what exactly does that entail?

In this chapter we will explain how we can think of this as a causal reasoning, where the user tries to connect the effect of bad program performance with the causes involved in writing a program. As we will show in Section 2.1, this involves a characteristic computer-assisted reasoning process. Our goal for Section 2.2 on page 8 will be to develop a set of abstractions that will allow us talk about this process more easily. Finally, in Section 2.3 on page 13 we will introduce causality theory, which will give us a better idea of how we can properly reason about causal dependencies.

2.1 The Task

At its very heart, performance analysis is a diagnosis task: We observe an undesired effect and want to find ways of fixing it. When a program starts using too much resources, the programmer is basically in the same spot as a doctor trying to figure out how to treat, say, a tummy ache: We have very good reasons to believe that the program or patient should be able to function normally *without* showing these symptoms. However, this does not mean that trying to suppress them directly is always the right idea. While throwing in pain medication or more resources might give temporary relief,

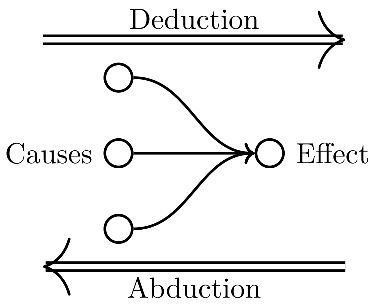


Figure 2.1: Deduction vs. Abduction

we might just as well be masking the real problem – and more gravely, end up producing new ones as an adverse side effect.

Therefore attacking the problem purely from the effect side is clearly short-sighted. We must find the *causes* - whether it be a bacterial infection or an inefficient loop. Once we have properly understood where the symptoms are coming from, we will have an easier time coming up with an appropriate solution, such as antibiotics or a more efficient data structure.

2.1.1 Reasoning

However upon close inspection, finding the “right” solution can be a strenuous mental task. Ideally we would like to use *deduction* to find the villain: Just take a hard look at the program and its inputs, and step through it in our head until we manage to deduce the symptoms. After all, resolving the mystery of a tummy ache might just happen to be as easy as remembering a heavy meal! However, this method quickly runs into problems where we lack a sufficient overview of the system’s inner workings. For medicine, grasping all mechanisms at work within a human organism is nothing but a distant dream. And equally the sheer volume of information produced by a program run will also quickly eclipse the capacity of the human mind. Where problems raise their heads from these depths, we need to employ more elaborate strategies.

Indeed our weapon of choice will not be deduction, but *abduction*[1]: the art of deriving causes from effects. As shown in Figure 2.1, this flips around the direction of our reasoning. This is a very good idea here because programs by their very nature build up complexity as they go along, processing their own data to produce ever more complex behaviour. It is therefore generally much easier to narrow down where an effect might be coming from than trying to predict all possible effects that a given program

[1]Not to be confused with abduction as in “Abduct a highly esteemed Haskell expert to solve the problem for us”. Even though effective, this method has been shown to scale badly.

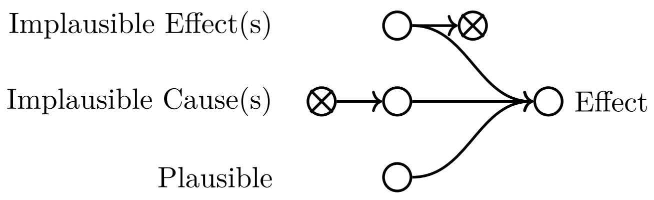


Figure 2.2: Abduction: Assessing Plausibility

element could have. Consequently, any method that allows us to climb the causal chain *backwards* will allow us to locate causes more easily.

How does abduction work? While deduction only makes statements that are true by construction, abduction is more heuristic in nature. Our starting point is the principle of sufficient reason, which states that nothing happens without a cause. This means that when we enumerate possible causes, we know for a fact that one of them must be true. To make progress we then filter this set by "plausibility" as shown in Figure 2.2: For some causes we might be able to deduce other effects, which by comparison to our observations can lead us to discard the cause as implausible. On the flip side, we can also use abduction recursively to show that only an entirely improbable combination of root causes could lead to the hypothetical cause[2]. Such plausibility checks will often substantially reduce the set of possible causes. For example, if a certain loop was at the heart of the problem, we would expect resources to be wasted there time and time again – just like causes for aches might be distinguished by checking for other symptoms. Ideally, cross-examination of all theories will yield us the "only possible explanation"[3] based on virtually nothing but a series of educated guesses.

2.1.2 Tool Support

Our task will be to help out the user with performing precisely this mental task. Note that we will *not* actually try to do abduction within our tool: In contrast to compiler optimisations the focus of profiling is explicitly to allow dealing with problems that are beyond the current scope of automatic program analysis. Instead, the task of profiling tools is "just" to offer the user exactly the information needed for facilitating the abductive reasoning task. Insight into the reasoning process will however help us to identify at what points we ought to provide assistance, and inform how we should present our information to the user.

[2]This principle is known as Occam's razor in scientific theory. There are a number of parallels to the black-box problem of natural sciences here, with the notable difference that our box is not actually black, but just very, very confusing.

[3]A phrase commonly used by Sir Arthur Conan Doyle's character Sherlock Homes, who still insists on calling the process "deduction".

Let us walk through the abduction process step by step. The first task in resolving a performance problem will be to assess the damage: Get a clear picture of what kind of resources get used – and therefore potentially wasted – in what manner. A profiling tool would support this step by allowing automated collection and classification of performance data. For example, at this stage we might point out that garbage collection seems to be taking a long time, or that we see a surprising amount of context switches. The more precisely we can characterise the problem's symptoms, the more focused our search for root causes can be.

Next, the programmer will have to form theories for the observed program behaviour. Without assistance, this is an exceptionally hard task, as just a quick read through the program source will most likely turn up ample potential candidates for any given kind of resource consumption. Therefore it is vital that we can cut down on the number of possibilities by identifying "hot spots" that seem to have a close causal connection to heavy resource usage. This is the part that is most commonly thought of as the "core" profiling task: Show program parts annotated with a descriptive break-down of the resource costs. Anecdotal evidence suggests that such statistics often radically reduce the amount of code that has to be considered in the first place: Similar to the Pareto rule-of-thumb, just 20% of the code generates about 80% of the cost[Fowler, 2002].

Such performance statistics can be used in several ways to generate plausible theories: Both constructively, by identifying consumption patterns that exceed expectations, as well as destructively, by noting the absence of certain performance characteristics. In order to help the user with this, profiling tools attempt to communicate patterns beyond simple causality to the user. For example, a standard profiling tool might point out not only that a certain function can be connected to a significant portion of the program's running time, but that there is actually a certain program *path* that leads to the hot spot. It is clear that the usefulness of the profiling tool will increase the closer we can get to actually suggesting and evaluating possible courses of action for the programmer.

2.2 Verbs and Nouns

Let us take a closer look at what a "complete" explanation would look like. Figure 2.3 shows a schematic: The story will start all the way back with the programmer's design decisions, then follow the trail throughout the source code, its compilation, optimisation and execution, and finally ends with the effects – in our case unsatisfactory resource usage. Full comprehension means that programmers can connect all these dots in their head, hopefully offering starting points for program improvements.

In order to get there, we have to communicate the nature of the causality graph. The causality network for a program run is a product of systematic program generation

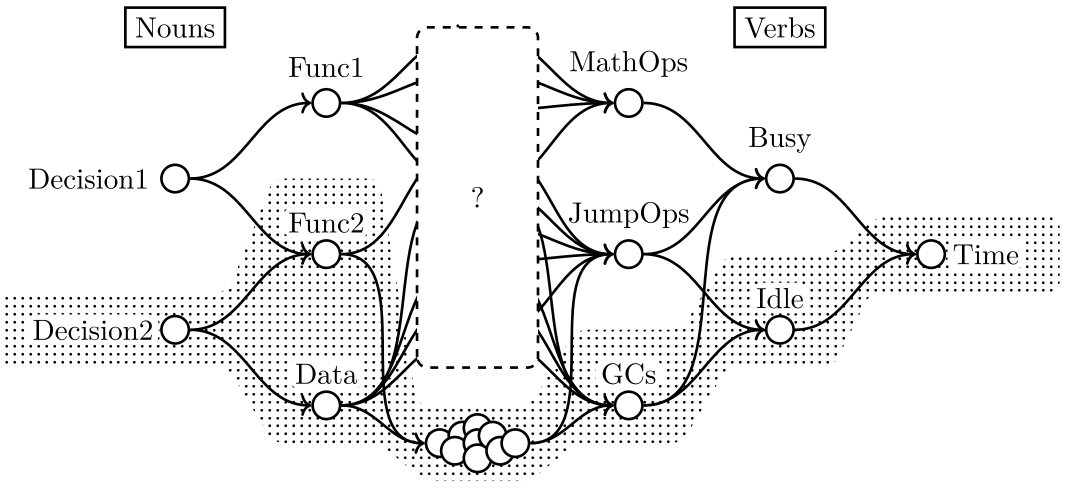


Figure 2.3: Schematic of an Explanation

and execution mechanisms, so the nodes have meaning that we could try to convey to the user. The closer we can get to the mental model of the programmer with this, the easier it will be for them to understand the causality network underlying program performance. Finding intuitive abstractions to talk about the causal processes is key. We will follow Irvin [1995] by calling causes close to the design decisions "nouns" and effects close to resource consumption "verbs".

2.2.1 Verbs

Verbs are an abstraction tool for reasoning about the "symptoms" of bad program performance. The "primitive" verb is simply resource usage, which might for example refer to time, energy or storage. However as explained, it is a good idea to consider intermediate causes for resource consumption instead in order to narrow the focus. This is especially true because modern programs execute on top of a significant stack of hardware and software, many with complicated performance semantics. For example, complexity arising from heap management is rarely the fault of the garbage collector, but actually of allocation and retention patterns within the program run. The better we can decompose the final performance into factors, the easier it will be for the programmer to influence the outcome.

Especially note that just raw resource consumption by itself is actually a rather poor indicator for spotting a performance problem. After all, clearly no amount of optimisation will ever reduce the run-time to absolute zero. Where we are looking for more subtle performance improvements, too much emphasis on resource consumption "hot spots" can actually become misleading: Productive work will end up overshadowing the inefficiencies.

Therefore decomposing the total performance is also a chance for us to specifically look for indicators that would not show up during normal operation: Just like a human will be able to identify certain symptoms as an "ache" or just "feeling uneasy", some types of program behaviour are a bad sign. For example, swapping significant amounts of data to the hard disk is usually a good indicator that there is an underlying problem, as there is basically no reason that a program should ever have to take advantage of such emergency measures[4]. Sometimes the user will even be able to make predictions about the kind of performance characteristics we should see: Discovering heavy memory consumption will be especially tell-tale where the program would actually suggest heavy number-crunching. The better we match verbs against the programmer's performance model, the more leads we have for the analysis.

Note that by construction many useful verbs will only reflect *parts* of the program's performance characteristics. This obviously means that focusing too much means we risk missing a performance problem altogether! For example, basing profiling on the amount of user-level CPU cycles might completely overlook the fact that the operating system spent most time swapping the program's memory to disk. On the opposite end of the spectrum, verbs also often end up overlapping wildly, especially if we consider verbs from different hardware or software "layers". From the CPU's point of view, a swapping operation will obviously involve a significant amount of, say, branch mispredictions, which might be another verb we might want to track. The programmer must understand these inter-dependencies in order to work effectively with verbs.

2.2.2 Nouns

On the other end of the cause-effect relationship, we have all influences that go into the program's execution. Such influences are not created equal: For profiling we will be most interested in nouns that represent something the programmer has the capability to change. It would be rather inappropriate to approach a performance problem by trying to manipulate input data or the inner workings of the compiler, for example. The main focus should be the impact of the programmer's *design decisions*, as we sketched back in Figure 2.3 on page 9.

This does not mean that outside influences are categorically not of interest for profiling. Quite the contrary – it is not hard to think of examples where the program's performance behaviour depends chiefly on, say, choosing the right library data structure or a script embedded within the input data. The real cause of the performance problem here is not the outside influence, but how the program interacts with it. Pointing out such connections helps focus the search, facilitating abductive reasoning as explained in Section 2.1.1. For example, if we spend a lot of time traversing lists we might consider

[4]Notable counter example: Varnish Cache, which uses swapping to speed up web applications.

using a different container type. On the other hand, if we could, say, determine that input data size does not impact performance we could rule out parsing code.

In the end, everything will most likely boil down to a change to the program, of which the source code is likely the representation most familiar to the programmer. Therefore, we should use the programmer's vocabulary as much as possible: Definitions and constructs the programmer wrote directly reflect their mental model of the program. We can refer to this model either by using names that already exist in the code, or by using for example line numbers to steer the programmer's attention into the right direction. This kind of lead should generally be enough for the programmer to be able to make the connection back to the design decisions that might be at the root of the performance problem.

2.2.3 Explanations

The whole point of verbs and nouns is that we can break down the performance problems in terms of simple abstract effects and causes. However, this simplification means that we can only ever penetrate the causality graph up to a certain depth. Even if we knew that a certain source code element causes a very specific performance problem – such as a stack overflow – this is not always enough to make it quite obvious "how" it all went down. At this point the programmer will need to be able to reason about the "intermediate" steps leading up to the performance problem as well.

Fortunately, at this stage in the analysis the programmer will most likely have settled for a certain aspect of program performance to focus on. This means that in contrast to verbs and nouns we do not need to limit ourselves to the user's mental model anymore. After all, talking about the program's inner working means reasoning about a lot of "incidental" information – such as how the compiler chose to apply optimisations, or how the CPU copes with our specific instruction mix. Not by chance, this is precisely the kind of information that the programmer normally aims to offload when using a high-level language. Yet if we want to talk about anything beyond the bare existence of a causal connection, we need to re-introduce the user to a portion of these implementation details.

This does not have to be too painful, as any experienced programmer ought to have some abstract notion of how the program gets compiled and executed. We can also ease understanding if we reference verbs and nouns familiar to the user as often as we can. For example, for profiling imperative programs it is quite common to link performance data to call stacks [Graham et al., 1982], which represent call hierarchies within the program. From our point of view, this is a simplified overview of the full causality network, as we are glossing over details such as what exactly caused a certain call to happen in the first place. In fact, this has even been seen to be a useful metaphor for functional programs, where the "true" control flow is often even more complicated [Sansom, 1994]. We can even extend this to parallel program, where it might be a good idea to break down exactly how and when threads were scheduled or preempted [Jones et al., 2009]. At the most extreme, Section 5.6.7 on page 158 will even advocate looking at intermediate language representations of our program in order to unpack its performance characteristics.

In the end, while more complex explanations can be unpredictable to the point of randomness, they dictate performance in broad enough strokes that involving them becomes indispensable once performance analysis reaches a certain point. And while we still expect explanations to only become relevant once we do a focused analysis, we should still aim to make our explanations as self-contained as possible. An ideal explanation "language" should be allow us to communicate exactly what we need to know in order to understand the causal dependencies, but nothing more.

2.2.4 Metrics

As the final corner stone, abductive reasoning needs an estimate for how "strong" causal connections are. After all, individual instances of resource usage are most likely not worth investigating, we are looking for patterns in the performance behaviour that are substantial enough to actually make a difference for overall program performance. This is where metrics come in: While verbs only identify undesirable events, metrics associate them with cost values, which we can measure and compare. For example, for Figure 2.3 we can easily map the verbs to quantifiable metrics such as total time spent, mathematical operations executed or garbage collection complexity.

We can use these metrics at multiple points when profiling. First, the user will need starting points in order to discover theories. Here statistics allow us to suggest plausible verbs, nouns and explanations. This is why we "profile" the program in the first place: Interesting causal processes should have large enough footprints that we can detect them in a program run. We will be much more willing to, say, pursue the theory that a certain function causes too much stack allocation if we can show that the function can actually be linked to a rather large amount of stack consumption.

Furthermore, these measurements come in useful when we want to gauge *relative* plausibility of theories. Investigating a certain function will look even more promising if we can show that no other function can be linked with a comparable amount of resource usage. On the flip-side, sometimes we can even show that certain theories can *not* be true simply because of the absence of certain traces in our data. As we will later see, there is in fact a good reason why collecting performance statistics is almost synonymous with the performance optimisation process: It allows us to spot even small problems within large and complex programs.

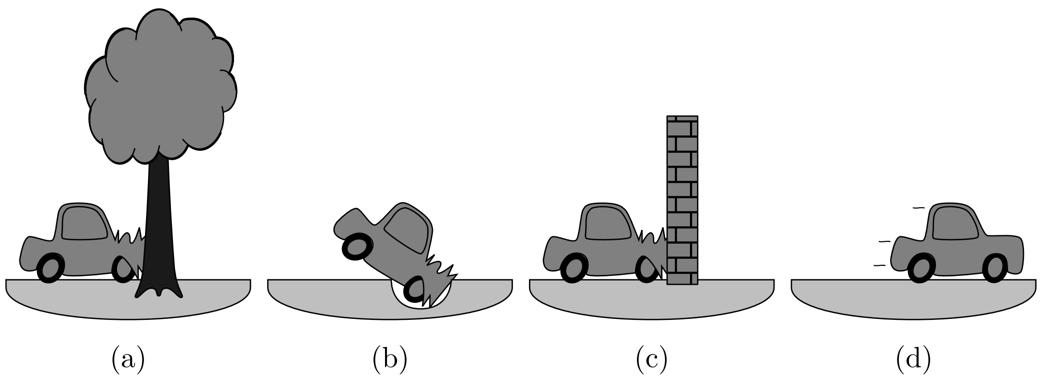


Figure 2.4: Causality Analysis of an Unfortunate Event

2.3 Causality

Up to this point we have used terms like "cause" and "effect" without proper definitions. This is not necessarily a problem, as humans generally develop a robust intuitive understanding of causality. However, our intuition can quickly go awry once we consider more abstract objects, such as the nouns or verbs discussed in the last section. To see why, let us consider a real-world example and attempt to explain our intuition. In Figure 2.4a we have depicted a car crash. If asked, we would probably say that the existence of the tree was one of the factors that caused the crash in the first place[5]. If we furthermore were asked to support that statement, we would most likely point out that if the tree had not been there, the car would just have driven by, as depicted in Figure 2.4d. But how sure are we of this argument? After all, we can not deny that there are many other scenarios (Figure 2.4b-c) that share both the property of the tree not existing *and* the car crashing!

Let us take a closer look. What we are doing is what Lewis [Lewis, 1973] describes as "counter-factual causality reasoning": We reason about a world where certain events did *not* happen. In this case, we could assign the symbol α to the tree's existence, and the effect ω would be the crash actually happening. According to Lewis, we now think about the *closest world* where α is false (the tree does not exist) and check whether ω becomes false as well. If that is the case, we are allowed to derive causation, which Lewis notates as

$$\neg\alpha \mathrel{\Box\!\!\rightarrow} \neg\omega$$

So according to Lewis, our loophole is that we regard the world depicted in (d) as *closer* to (a) than either (b) or (c). But what does "closeness" mean then? From a practical point of view, our choice can seem down-right arbitrary. After all, a woodworker might tell us that getting to (b) would just require us to uproot the tree, while manufacturing

[5]Even though, as was noted, blaming the tree for these events seems decidedly unfair.

scenario (d) would probably involve levelling the ground afterwards. On the other hand, keeping our physical education in mind we might argue that from these options scenario (c) might have the best chance at upholding the law of conservation of mass.

And as it turns out, Lewis' formalism does not actually offer too much help in this regard. Technically we could actually use either way of measuring closeness. Worse: Depending on the kind of scenario under consideration, it might actually make sense. This is a well-documented weakness of Lewis' theory, and apparently quite hard to solve conclusively. The best we can generally do is to impose a structured model that formalises our intuition [Pearl, 2000; Taylor, 1993]. Consequently we will have to make our intuition concrete in one form or the other at this point, and with Lewis' theory we will have this choice front and centre, without prematurely locking us into a certain thought model. As a result, we will generally stick to Lewis' nomenclature even where we were influenced by later work on the topic.

2.3.1 Context

What seems to steer our intuition is the *context* of the scenario. After all neither trees, walls nor holes normally occur where cars are driving! Therefore to subtract the tree from the scenario, we instinctively try to compensate by moving towards what we see as the "default" state. This default can change: Just imagine that we knew that the tree was planted specifically to provide a barrier, scenario Figure 2.4d might suddenly appear a lot more "normal" than before.

What we want is consistency: It is easy to postulate an event not happening, but if we remove a tree, we can not simply assume that there is "nothing" there. We have to assume that something fills the void. These replacement events should cause as little damage as possible to what we perceive as the consistency of the situation. Just like with abduction, we can view this from two sides:

1. **Cause consistency:** It should be plausible that the scenario has come to be by just some minor "miracles" in the past.

   This is essentially the "woodworker" line of reasoning from the last section: If a wondrous fairy would have to work too hard to get to the situation, it is probably not the most plausible alternative. Ideally we would like to simply exchange exactly one event by a related event.

2. **Effect consistency:** Any new effects should be as plausible as possible in the context of the original world. As mentioned, we might for example challenge this if nothing being there might lead to an unnatural situation.

   Note that this can be formulated as already covered by the first criterion: We assume an intelligent or otherwise purposeful entity in the past that would try its

> best to retain the effect in question. Our "miracle" would then be exerting the right amount of pressure on that entity to make it change the actual implementation. This applies strongly to profiling, as we have a guaranteed purposeful entity in our causality network: The programmer!

Note that this definition is somewhat circular: We define plausible causality in terms of the causal network it generates. Therefore this is, again, a consistency requirement that we need to instantiate on a per-case basis. Especially note that a "plausible change" does not always have to be a small change: If the car in question was carrying a person on her way to starting World War III[6], alternate world events might end up completely different. Yet the new scenario would clearly still be "plausible". As we will see later, we will generally favour small and predictable miracles, even if it means that we need to stretch effect plausibility a bit.

2.3.2 Application to Programs

Let us return to our main objective, which is reasoning about program performance. In contrast to the example in the last sections, we have to deal with abstract rather than physical objects: Our nouns and verbs will be program elements, runtime constructs or cost statistics. We will later see that we can express all of these as *languages* of some form, so let us have a closer look at how we can reason with them.

Suppose we have the following implementation of the factorial function:

```
1  fac :: Int → Int
2  fac n = foldr (*) 1 [1..n]
```

Listing 2.1: Example Haskell Program

This Haskell program implements the factorial function in a straightforward way: Conceptually we request an enumeration of numbers from 1 to n using the [1.. n] syntax, which we proceed to multiply using the foldr higher-order function. We will see later that even this simple program has quite complex compilation and execution. For the moment, we only have to know that it has, in fact, a performance problem: In comparison with more efficient versions its execution consumes excessive amounts of stack space due to recursion behaviour of foldr.

So our "root" noun here is the decision of the programmer to use foldr, and probably some of the assumptions that went into that decision. However, a profiling tool can not look into the mind of the programmer, therefore we use the next closest thing: We observe the fact that " foldr " was used in line 2 of the listing. To help diagnose the

[6]Allegedly careless use of unsafePerformIO played a role.

performance problem, we would like to connect this cause to the effect of excessive stack consumption. Put in terms of causality, we have two events

$$\alpha \equiv \text{Use of “}\mathsf{foldr}\text{” in line 2}$$
$$\omega \equiv \text{Excessive stack usage}$$

for which we want to decide $\neg\alpha \mathrel{\Box\!\!\rightarrow} \neg\omega$. Or in words: Would removal of “ foldr ” possibly fix our performance problem? And like in the examples we again bump right into the “closest world” question – how would a comparison program without “ foldr ” even look like?

2.3.3 Alternate Worlds

We have a few options. We could play dumb for a moment and simply lexically remove the expression, not entirely unlike uprooting the complete tree like in Figure 2.4b:

```
fac :: Int → Int
fac n = (*) 1 [1..n]
```

Listing 2.2: Faulty Alternative

This is a rather useless suggestion, as the new “program” is not well-typed, and therefore simply invalid. Even if we could get it to pass the type-checker, we have significantly changed how program elements interact: Now we actually apply the (∗) function instead of passing it as a parameter! Clearly this makes for a poor point of comparison.

On the other hand, we can also try to be “smart”:

```
fac :: Int → Int
fac n = foldl (*) 1 [1..n]
```

Listing 2.3: Smart Alternative

Now we have simply replaced “ foldr ” by its sibling function “ foldl ”. As far as distance to the original goes, this might seem like an excellent idea: We retain not only type-correctness (at least for this example!), but we actually get the same functionality while just changing around the concrete implementation. This is probably one of the options the programmer will be thinking about once she has managed to track down the problem. However, for our purposes this is actually *too* smart, as the new code now could have a *different* performance problem[7]! We have essentially run into the trap of constructing Figure 2.4c: By being too cautious about side effects, we have ended up reproducing the very effect we want to track.

[7]Even the recent work of Breitner [2004] cannot always prevent this, see e.g. Section 6.2 on page 190.

2.3.4 Minimal Change

So what would be our best equivalent to the unimpeded car in Figure 2.4d then? We want neither implausible causes nor effects, which for us means:

- disturbing the language syntax tree by actually removing the element or
- introducing new behaviour by substituting meaningful code.

Put together, this leaves us no choice but to punch a "hole" into our program:

```
fac :: Int → Int
fac n = □ (*) 1 [1..n]
```

Listing 2.4: Alternative with a Hole

The hole symbol □ acts as a place-holder for the removed expression. In fact, in order to not miss any possible effect we would like the new program to have no behaviour that depends on the original " foldr " expression. We could for example approximate this in a real Haskell program by setting "□ = undefined". This would make the program terminate on the first actual usage off "fac". That might appear rather extreme, but among all of its effects, the new implementation is guaranteed to never have a performance problem if " foldr " was to blame for it! This is exactly the kind of property we need to reason reliably about causality.

But is substituting hole values the only way to find alternate worlds for causality analysis? After all, this might easily lead us to over-estimate the ultimate effects of an expression. As explained, context is important to consider, and we might occasionally find that there *are* ways to remove the expression without having to involve place-holders. For the sake of argument, let us add some error checking to our factorial function:

```
fac :: Int → Int
fac n | n < 0 = error "Factorial␣undefined!"
      | True  = foldr (*) 1 [1..n]
```

Listing 2.5: Example with Error Checking

To determine whether this new check has actually introduced the performance problem, comparison with the pathological "fac n = □" alternative would probably not be the best course of action. After all, we could instead simply eliminate the extra branch and compare with the original program from Listing 2.1 instead. This would give us the correct answer: The performance problem occurs in both cases, therefore the check is clearly not to blame! We will later see that in a few cases we can actually do something like this systematically.

To summarise, there are parallels between reasoning about causality in the real word and handling it for abstract languages. If anything, it is actually easier to define what the "context" of a language construct should mean, and the hole term gives us a good default method for removing program elements.

2.3.5 Transitivity

Causality thus far has been purely about whether or not we can infer a causal connection between a singular cause-effect pair. However, before that point we have been using the term of causal "networks" as our metaphor for thinking about performance problems. How do we need to extend our reasoning to allow us to cover these?

It is quite clear that we want causal transitivity: If we have a cause α, another cause β and an effect ω, we would expect that

$$(\neg\alpha \mathrel{\Box\!\!\rightarrow} \neg\beta) \land (\neg\beta \mathrel{\Box\!\!\rightarrow} \neg\omega) \Rightarrow (\neg\alpha \mathrel{\Box\!\!\rightarrow} \neg\omega)$$

which counter-factually states that if α was to be false, we would see both β as well as ω becoming false. The reasoning appears sound, as after all ω depends on β happening, which in turn depends on α. This is how we would commonly think about a cause network: Every effect is caused via transitivity by every connected cause preceding it.

However as we should have seen by this point, we should not take consistency for granted when reasoning about causality. If we return to our car crash example, let us name the car crash ω, the tree's existence β – and the decision to plant the tree in the first place α. Normally we would expect α to cause ω now, but what if the tree was planted to prevent the ground from slipping? Again, reversing α would still see ω become true, albeit this time because the ground has eroded away over the years! At heart, this is quite related to our issues with finding good close worlds. When we considered causality between β and ω, we made no assumptions about why β became false in the first place. We assumed a small "miracle" and restored consistency from there. However here the prior change invalidates our assumptions: In the new alternate world entirely new outcomes become plausible.

How are we going to handle this? There are basically two ways. First off, we can shift around at what stage we look for causes and effects respectively. If for example we only consider causes for the car crash that are in close temporal proximity, we will only find β, but overlook α. However, if we take a step back and take the full history into account, we find that it is much more worthwhile to "short-cut" our reasoning by directly considering causality between α and ω. We will later see that this approach can indeed be used to eliminate spurious causal relationships entirely.

On the other hand, attempting to identify such instances can increase the complexity of our task enormously. After all, in the example we could ask tricky questions like whether changing the original decision might not – via complex mechanisms – end up causing the car to not attempt this particular trip in the first place. Also consider the program we considered in the past sections: A hypothetical decision to not use `foldr` would most likely not result in the "hole" program from Listing 2.4. So some of the effects we would associate with it would, in fact, not be "true" effects. Yet even knowing this; the complexity of the program design process means that we have basically no way of predicting what the "plausible" alternative is going to be.

However, this was to be expected. After all, if we could perform a "perfect" causality analysis, we would be able to connect design decisions with resource usages all by ourselves, therefore solving the performance problem. Fortunately, this is not actually our task. For supporting abductive reasoning all we need is the ability to point out and evaluate *possible* causal links. Overestimation at this point merely means that we are subjecting the user to more false positives. While this is clearly undesirable, it comes with the nature of our task.

2.4 Conclusion

At this point we have developed a clear picture of what the profiling task is all about: Assist the programmer in finding causality links between the influences on program execution and the final program performance. We have classified concrete causes and effects into abstract causes (nouns), effects (verbs) as well as intermediates (explanations). We then explained how we can establish causality between these entities. To recapitulate, we are allowed to view something as an effect of a cause α exactly if:

1. It does not happen in an alternate world where we miraculously invalidated α
2. or – if it turns out that we cannot decide the first criterion – it is indirectly caused by any of the other effects of α.

It is clear that the concrete implementation of our profiling tool will have to vary heavily depending on the kinds of nouns and verbs we ultimately want to reason about. Not only will we have different verbs, nouns and explanations, but the way we reason about them causally will also heavily depend on the development context. For example, reasoning about the performance of an imperative language would require a substantially different analysis – and therefore tools – than dealing with functional programs. In the following chapter we are going to instantiate all of these concepts for our work.

Chapter 3

Haskell

"What is up? I can like flowers even if I am a barbarian, right?"

Asterix and the Goths, René Goscinny & Albert Uderzo

Our focus for this work will be to develop a new general-purpose profiling solution for code written in the programming language Haskell [Peyton Jones, 2003]. We have a good motivation for limiting ourselves to a single language: As explained in the last chapter, if we want to be effective in supporting abductive reasoning, we need to know as much as possible about the programmer's mental models. And the programming language is undoubtedly one of the most important factors shaping the user's mental image of the program's functionality.

This is especially true for Haskell, as it proudly breaks with venerable software development traditions by focusing entirely on purely functional programming with non-strict semantics. As a result, Haskell programs often end up looking entirely differently from programs written in more conventional languages. This goes beyond simple syntax: It is not uncommon for learning programmers to voice sentiments along the lines of "Haskell is changing my brain", or that it feels like re-learning programming entirely. For our purposes, this makes it plausible that there will also be unique mental models on the programmer's part, justifying a specialised treatment of the development environment.

Furthermore, for an arguably niche programming language Haskell is in the unique position of having a healthy community of "real-world" software development. In fact, functional programming has become main-stream enough that the International Conference on Functional Programming regularly hosts special forums to allow commercial users to discuss the practicality and – indeed – performance of a wide variety of language use cases. It is not uncommon for Haskell to take centre stage on these occasions, as its mix of high-level programming combined with efficient implementations has proved to make it remarkably good at tackling problems from the real world.

In this chapter, we explain the particular issues we face with profiling Haskell code. We will start with giving a general overview of the language in Section 3.1, which will inform our priorities for the profiling solution in Section 3.2. To get an idea about the causal inter-dependencies, we will then proceed to review the Haskell infrastructure. Given the nature of compiler-based programming language implementation, this will be split into two parts: Section 3.3 on page 26 will serve as an overview of the compilation pipeline of the Glasgow Haskell Compiler, with Section 3.4 on page 30 demonstrating how its transformations work for an example program. In Section 3.5 on page 41 we will then explain the execution of Haskell programs, and derive an abstract performance model.

3.1 The Language

It is a widely held belief amongst proponents of advanced programming methodologies that programming at a high level will lead to better programs. The logic is that the farther we can remove the programmer from the idiosyncrasy of the hardware platform and the necessity to write noisy boilerplate code, the more brain-space she can dedicate on solving the problems that actually make programming hard: Truly understanding the problem requirements to the point where we can decide on a program design that minimises cognitive effort at every step along the way. This approach, conventional wisdom suggests, leads to software that is both effortless to write as well as maintain.

In practice, this basic idea has inspired a zoo of programming languages to spring into existence, the size of which might actually surprise outside observers. The trouble is that it is surprisingly hard to build programming languages that have clear meaning, can be written in a compact way, *and* fit our mental models well. What seems to be perfectly intuitive for one application can seem bewildering and forced for another. As a result, to this day we have not only seen the birth of many programming languages, but also have whole families and paradigms vying for the crown of being the most "expressive". It stands to reason that this contest will never truly be decided.

3.1.1 Purity

In the ongoing evolution of computer language design Haskell occupies quite a unique spot. At the present time most main-stream programming languages still rely mostly on the *imperative* approach, which uses the metaphor of manipulating some notion of computer-internal state to describe algorithms. This matches both the reality of programming on the hardware level as well as the practice of interacting with anything outside the computer. In fact, there are situations where structuring programs in terms of changing state is without much doubt the most natural approach.

So it might appear puzzling that Haskell instead adopts the *purity* ideal: Describe as many computations as possible in a way that does not depend on state manipulation of any kind. For example, we saw in Listing 2.1 on page 15 that we can describe the factorial function as using a fold over a list: a pure computation involving no side-effects at all. Compare this with a typical imperative implementation:

```
int fac (int n) {
    int x = 1;
    for (int i = 1; i <= n; i++)
        x = x * i;
    return x;
}
```

Listing 3.1: Factorial in C

We see that idiomatic imperative programs uses state, such as the counter variable `i` or the accumulator `x`. Experienced programmers will find this easy to read, as they can easily play through different loop states in their heads.

Yet Haskell actively *discourages* this kind of programming. Beginning Haskell programmers are generally split between bafflement and curiosity when they learn that there is simply no straight-forward equivalent to the above implementation in Haskell. Our closest choices would basically be to either encode the loop state as parameters to a recursive helper function, or involve libraries [Launchbury and Peyton Jones, 1994] that emulate imperative paradigms. This will allow us to use to write small imperative programs such as the one cited above. However the more we want to share the state between functions and procedures, the more of a fight the language will put up: We will be forced to explicitly state our intentions in the type of every function that takes part.

3.1.2 Higher Order Programming

Of course there is a good reason why Haskell goes to great lengths to promote purity. The basic idea is that while pure functions might appear harder to write at first, they become more straightforward to *use*. In a purely functional language, we basically know that the return value of a function of type "`Int →Int`" will depend exactly on the sole parameter we pass in. It does not matter how often, in what order or in what program phase we call the function: Its behaviour will always be the same.

This property might not seem particularly remarkable, but it is in fact very useful if we consider the context of development of large applications. The reason is that it forces the programmer to think carefully about introducing data dependencies. This will not only make it "physically" harder to write hard-to-understand programs, but also enables advanced programming techniques that would otherwise be too unsafe to use. Take for example higher order programming, where we use functions as first-class values and encourage writing programs by combining them. For example, the foldr function we used in Listing 2.1 is defined like follows:

```
foldr :: (a → b → b) → b → [a] → b
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Listing 3.2: Definition of foldr

This function implements a very basic idea: Starting from certain value, we iteratively "fold" all values from the list into it. As Listing 2.1 demonstrated, getting to the factorial function from this point is as simple as setting f to the plus operator, use 1 as the starting value and an enumeration of numbers as the list.

So in the end, arriving at the correct result depends on two functions co-operating: foldr and the plus operator. Neither function knows anything about the other until we choose to combine them. Without purity, this might be somewhat tricky: Are we sure the two functions will not interact? Will they get called in the order they expect? It is not hard to see that for larger examples, these sort of questions might become highly non-trivial to answer. In the end, we might even be tempted to specialise interacting functions to fit each other, generating treacherous inter-dependencies between software components that could have been independent.

3.1.3 Optimisation

Despite the learning curve, higher order programming with purely functional code is a great match. But for the purpose of this thesis we have to especially acknowledge the performance angle: Implemented in a straight-forward way, programs using an imperative style will often turn out to be substantially faster. The reason is simply that the higher-order style promotes combining together a high number of general-purpose functions. The required "glue points" between functions will make assumptions about the data format of the passed data. It is not hard to see that these might not always coincide with the most efficient choice.

For example, the foldr function from Listing 3.2 has to be able to work with any Haskell function that might get passed as f. Therefore a direct implementation would perform one indirect jump per list element at minimum, even if the only code ever called was the plus operator. Compilation of our imperative version would see the hard-coded plus operator at this point, and simply short-cut the jump. Even more gravely, the Haskell way of folding over a list seem to require a list to exist in the first place. The imperative implementation directly uses a light-weight loop at this point.

However not all hope is lost. In being less specific – or more declarative – about the implementation, we allow the compiler more freedom for performing *optimisations*. For example, given the compact size of the foldr definition, it is cheap to just copy the code and specialise it for our purposes. This would trivially restore our ability to in-line the plus operation in the last example. We can even teach the compiler to recognise and eliminate inefficient intermediate data structures where we find them. For example, we can generalise the enumeration of numbers as follows [Gill et al., 1993]:

```
eftInt :: Int → Int → [r]
eftInt n to = build (λstep start → eftInt step start n to)
eftIntFB :: (Int → r → r) → r → Int → Int → r
eftIntFB step start n to = go n
  where go n | n >= to     = start
             | otherwise   = n `step` go (n+1)
build :: ∀ a. (∀ b. (a → b → b) → b → b) → [a]
build g = g (:) []
```

Listing 3.3: Fusion-enabled Enumeration

This slightly simplified implementation of eftInt splits its task into two components: An abstract function for generating numbers (eftIntFB) and a function that parameterises it to produce the expected list (build). The key observation is that now foldr is actually "dual" to build, allowing for the following transformation:

```
foldr step start (build g)  ⟹  g step start
```

Listing 3.4: Short-cut Fusion Rule

In the example of our tried-and-true factorial function, this rule actually allows the compiler to eliminate the intermediate list and "fuse" the number creation with the calculation of the factorial. We will look at the details of this process in Section 3.4. At this point, the critical observation is that this transformation is only valid because we know all components to be pure functions. If we had to assume possible inter-dependencies between functions f and g in the above rule, arguing for its correctness would be much harder.

3.2 Objectives

For performance the programming style that Haskell promotes can be a blessing and a curse at the same time. On one hand, it is demonstrably true that Haskell programs can perform as fast as comparable programs written in imperative languages. Especially fusion techniques can be powerful enough that even the performance of low-level languages can be "beaten" [Mainland et al., 2013]. But on the flip-side, the declarative style offers little in terms of performance *guarantees*. Whether a program achieves high performance or ends up crawling along at snail's pace is mostly a matter of whether the relevant program optimisations found their chance to shine.

This can lead to problems for real-world programming scenarios. Experienced programmers will obviously try to make use of the optimisation facilities given to them, and write their programs in a way that they expect to optimise well. For example, a programmer might opt to restrict herself to use functions with well-defined short-cut fusion rules [Gill et al., 1993], and proceed to reason about the program performance with deforestation as the expected default. However, this is dangerous, as it is easy for unforeseen interactions to sneak their way into the process. Suppose we tried to compute the product as well as the sum of a list of numbers at the same time:

```
test :: Int → (Int, Int)
test n = (foldr (*) 1 nums, foldr (+) 0 nums)
  where nums = [1..n]
```

Listing 3.5: Sharing Hindering Rules

The shared [1.. n] list is a hard nut to crack for the compiler, as it has to decide whether the cost of duplicating the work of enumerating the numbers is outweighed by the benefit of being able to apply the rules. Here it would actually fall back to the more conservative choice of skipping the rules, subverting the programmer's operational expectations. To make matters worse, sometimes such sharing can appear without the programmer's knowledge, and is even quite hard to prevent [Breitner, 2012]. Once code complexity reaches a certain point, these kind of interactions can degrade predicting performance to a high-level guessing game. In fact, polls among Haskell users consistently results in performance unpredictability being cited as a primary concern [Tibell, 2011].

Our goal will be to solve this problem from the profiling side. Our main hypothesis is that real-world purely functional programming with aggressive optimisations is here to stay, and that we can make it more workable by allowing users to trace more easily how their program design decisions led to the observed run-time characteristics. As we will see, program optimisations make this especially challenging, as they substantially increase the complexity of the causality network.

3.3 GHC Overview

Before we can start fleshing out how our profiling solution ought to work, we first need a robust understanding of what we are up against. Our primary concern is the inner workings of Haskell compilation, and especially how it shapes the performance of

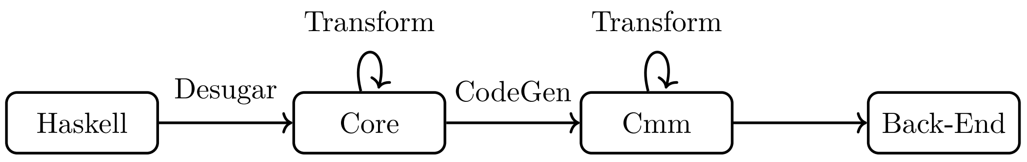


Figure 3.1: Glasgow Haskell Compiler – Compilation Pipeline

the compiled program. As sketched out earlier, from the compiler's point of view the unconventional language design choices of Haskell represent a number of challenges as well as opportunities. Compiler passes, libraries, as well as the runtime system should be tailored to make maximum use of the language's characteristics.

Given these challenges, it is noteworthy that with the award-winning and allegedly glorious Glasgow Haskell Compiler GHC [Peyton Jones et al., 1993] we have a compilation infrastructure that has managed to build up a reputation for being industry-grade and fit for real-world programming. It will be our primary point of reference for the analysis, as well as our target for implementation. Other notable compilers under active development include the Utrecht Haskell Compiler [Dijkstra et al., 2009], the Jhc Haskell Compiler or the Intel labs Haskell research compiler [Liu et al., 2013]. The main differences between these compilers concern the exact nature of the used intermediate languages, such as GRIN [Boquist and Johnsson, 1997] for UHC and Jhc. However the transformations performed on the program representations share common themes, and we would expect our analysis to adapt to any given compiler infrastructure in a straightforward way.

3.3.1 Core

Figure 3.1 shows a high-level overview of the GHC compilation pipeline. As customary for compilers, the program gets translated through a number of intermediate languages. Given their role in compilation, the primary design goal of these languages is to facilitate transformations. For this it is paramount to be as general as possible while maintaining a clear understanding of the performance implications of their changes. Consequently, the design of intermediate languages differs notably from actual programming languages in that they favour simplicity in design and a clearly defined performance model over brevity or ease of use.

For GHC, the principal intermediate language is called *Core* [Peyton Jones and Lester, 1992; Santos, 1995]. In comparison to the Haskell source language, it is a rather simple functional language with its theoretical roots growing down all the way to System F [Girard et al., 1989], an extension of lambda calculus with types. GHC extends this base language further by introducing primitive types and first-class `let` and case expressions. This makes the program representation just low-level enough to not limit transformations, while maintaining strong theoretical roots [Peyton Jones, 1992] at the same time. After all, even with these extensions the language is still extremely light-weight with only 6 distinct expression forms (ignoring types):

```
data Expr b
  = Var    Id
  | Lit    Literal
  | App    (Expr b) (Arg b)
  | Lam    b (Expr b)
  | Let    (Bind b) (Expr b)
  | Case   (Expr b) b [(AltCon, [b], Expr b)]
data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))]
```

Listing 3.6: Core Definition (Without Types)

This simplicity is why translating the parsed and type-checked Haskell program into Core is called "desugaring": We remove all syntactic sugar and break the language down to its most basic components.

Our running example, the factorial function from Listing 2.1, would look like follows in pretty-printed Core:

```
fac :: Int → Int
fac = λn → foldr (*) (I# 1) (enumFromTo (I# 1) n)
```

Listing 3.7: Core Example

The desugared program shows a number of differences: Instead of using a parameter pattern match, fac is now defined using an explicit lambda expression. Integer constants now appear in the "boxed" form, using the magic I# constructor to make them non-primitive. Finally the enumeration syntax was simply replaced by the library function enumFromTo.

3.3.2 Types

Due to its System F heritage, Core will treat types like first-class values, passing them around as parameters and variables where the type of an expression cannot be determined at compile time. These type terms have no run-time representation, and will simply get eliminated at the end of the Core transformation stage. As we are mainly concerned with profiling of run-time costs, this means that we will pretty much ignore types for the rest of this document. However, strictly speaking this means losing performance-relevant causality information, as the choice of type can absolutely have an impact on performance.

Consider the following program:

```
square :: Int → Int
square n = n ^ 2
```

Listing 3.8: Type Performance Problem

We would obviously expect this function to be very efficient. Yet in fact, for a number of GHC releases the compiler did not only miss the obvious `n*n` replacement, but also made a very poor choice on the type level. To understand how, let us look at the Core code after desugaring, this time with types:

```
square :: Int → Int
square =  λn → (^) @Int @Integer $fNumInt $fIntegralInteger
                    n (__integer 2)
```

Listing 3.9: Type Performance Problem – Core

Terms preceded by `@` are the type terms we were talking about: As the (^) operator has type (Num a, Integral b) ⇒ a → b → a, System F demands that we "pass" the concrete type to use for a and b, as well as two Num and Integral type class dictionaries [Hall et al., 1996] to be able to use them[1]. The important point is that the Haskell compiler went for the conservative default choice of using the Integer type for b. This type is normally meant for operating on arbitrarily large numbers. A rather unfortunate choice, as we know that the largest number encountered will in fact be 2!

However, we can probably assume that these sort of problems are rare. Even where we run into them, it is not too unlikely that the *value* in question will have as much of an influence on the result as the *type*. In our example, a good profile might be able to pin the performance loss to the "2" literal and Integer primitives, which is enough to suggest the right connection[2]. This is rather fortunate for us, as the troubles with accurate type error reporting have shown that reasoning about causation with type systems can be a significant challenge (see for example Stuckey et al. [2004]).

3.3.3 Cmm

As we will see later, Core is flexible enough to express all of the most important code transformations while in essence retaining the functional nature of the original Haskell code. However eventually GHC will have to break out of this abstraction and deal with the realities of adapting the program for sequential evaluation. To this end, GHC transforms Core into the high-level portable assembly language Cmm (related to `C--` [Peyton Jones et al., 1998]) at this point.

[1]Dictionary terms carry a `$` prefix in our example. In contrast to types, dictionaries correspond to function tables at runtime, so we think of them simply as a data structure containing function values.

[2]Which is, in fact, how this problem was discovered in the first place.

This intermediate language acts as a middle-ground between the functional transformation pipeline and the assembly generation in the compiler back-end. This is where GHC will decide on the concrete stack layout, register allocation as well as interactions with the run time system. Furthermore, there are some code transformations as this point, such as common block elimination or block concatenation. However, we see these optimisations as relatively straight-forward local optimisations, with limited influence on the program's performance. Therefore, for the coming sections we will simply pretend that the program is going to be a "direct" translation of our Core code. We will however revisit Cmm in Section 5.5 on page 143 to discuss implementation issues.

3.4 Transformations Example

To get a feel for code transformations at work, let us take a closer look at how GHC would translate the factorial function from Listing 3.6. The optimisation process would start with the desugared Core shown in the listing, and proceed to apply (presumably) beneficial transformations. We only stop once we are reasonably sure that we have exploited most opportunities for improving the program.

For our example, we might start with replacing the general enumFromTo and (∗) functions by their specialised counter-parts, eftInt and timesInt:

```
fac = λn → foldr timesInt (I# 1) (eftInt (I# 1) n)
```

Listing 3.10: Core after Specialisations

This corresponds to eliminating types and dictionaries as explained in Section 3.3.2. Note however that foldr is still polymorphic, and cannot be specialised. This is simply because it does not care about the element type at all, and therefore does not need specialised versions. We would have to in-line it in order to exploit our knowledge about types. Yet as we shall see, we can gain even more performance by applying rules.

3.4.1 Rules

The idea behind user-define rule transformations [Peyton Jones et al., 2001] is to match and replace certain local patterns within the code. The powerful property of this system is that Haskell libraries can feed it with custom transformation rules that exploit the specific properties of their data structures and functions. For example, the definition of eftInt mentioned in Listing 3.3 actually appears as a rule in GHC.Enum:

```
{-# RULES ''eftInt'' [~1] ∀ x y.
      eftInt x y = build (λstep start →
                                   eftIntFB step start x y) #-}
```

Listing 3.11: eftInt Rule

The left hand side of the equation clearly matches our program with $x = I_{\#}\ 1$ and $y = n$. If we paste the right hand side of the equation into our program and replace x and y by their assigned values, we obtain:

```
fac = λn → foldr timesInt (I# 1)
            (build (λstep start →
                       eftIntFB step start (I# 1) n))
```

Listing 3.12: Core after eftInt Rule

At this point, the rule achieved little more than a slightly more controlled in-lining. However its main function is to ensure that *other* rules fire correctly. Namely, at this point we can apply the rule from Listing 3.4, originating from GHC.Base:

```
{-# RULES "fold/build" ∀ k z (g :: ∀ b. (a→b→b) → b → b).
        foldr k z (build g) = g k z #-}
```

Listing 3.13: fold/build Rule

We can instantiate this rule with k = timesInt, $z = I_{\#}\ 1$ and g the lambda term we got from the eftInt rule.

This will "fuse" foldr and build:

```
fac = λn → (λstep start → eftIntFB step start (I# 1) n)
            timesInt (I# 1)
```

Listing 3.14: Core after fold/build Rule

Now we can simply β-reduce the term and obtain:

```
fac = λn → eftIntFB timesInt (I# 1) (I# 1) n
```

Listing 3.15: Core after Rules

which is a clear improvement: The build / foldr duality has allowed two rules from different library paths to collaborate in a structured and arguably predictable manner. As far as local code transformation go, our code is now completely simplified. The only way forward is now to unfold eftIntFB.

3.4.2 Basic Floating

For unfolding the definition we need the Core code for eftIntFB. Normally we would have this available at this point, as GHC would save unfold-able pre-optimised definitions along with the compiled library. However for our purposes let us take the small detour and reconstruct the Core by hand by starting from the definition of eftIntFB. To recapitulate, here is again the definition we gave in Listing 3.3 on page 25:

```
eftIntFB :: (Int → r → r) → r → Int → Int → r
eftIntFB step start n to = go n
  where go n | n >= to     = start
             | otherwise   = n `step` go (n+1)
```

Listing 3.16: eftIntFB in Haskell

Which yields us the following Core:

```
eftIntFB :: ∀ r. (Int → r → r) → r → Int → Int → r
eftIntFB = λstep start from to →
  let go = λn →
        case (>) n to of
          True  → start
          False → step n (go ((+) n (I# 1)))
  in go from
```

Listing 3.17: eftIntFB in Core

The effects of desugaring are a bit more striking this time: The where clause from the original Haskell code becomes a let expression, and the pattern guard was converted into an equivalent case expression. Note that we are again dealing with a polymorphic function, so for full Core we would also pass the type @r around explicitly. This part is however not critical for performance or optimisations, so we will again ignore it.

Let us start optimising this fragment. To get started we can specialise and unfold the definitions of the (>) and (+) operator to the Int type. This exposes the primitive numerical operations:

```
eftIntFB = λstep start from to →
  let go = λn →
        case (case n of I# n# →
              case to of I# to# →
              (>#) n# to#) of
          True  → start
          False → step n (go (case n of I# n# →
                                  I# ((+#) n# 1)))
  in go from
```

Listing 3.18: eftIntFB after Specialisations

Unfolding the operators in place has left us with a number of case statements for “unpacking” the boxed integers. This often means overhead, as it is generally better to directly use primitive values whenever we can. In our case, the code is clearly not very efficient: every loop iteration will unpack "to" again, and n will even get unpacked twice if we follow the False branch.

Fortunately, we can easily fix this by "floating" the case statements outwards:

```
eftIntFB = λstep start from to →
  case to of I# to# →
  let go = λn →
        case n of I# n# →
        case (>#) n# to# of
          True  → start
          False → step n (go (I# ((+#) n# 1)))
  in go from
```

Listing 3.19: eftIntFB after Floating

Generally speaking, selecting the right position to put a binding is a non-trivial endeavour [Peyton Jones et al., 1996]. For unpacking operations however, we can stick to the rule of thumb that we want to push them as far to the outside as possible. For the $n_\#$ binding, this means we have to stop at the lambda where n gets introduced, but $to_\#$ can actually float all the way up to the top level of eftIntFB.

3.4.3 Worker/Wrapper Transformation

As result of the floating optimisation, the unpacking of n is now actually the first thing go does when it is invoked. This begs the question: Why pass a packed integer number in the first place if it will get immediately unpacked? If the calling code happened to be operating on unpacked integers in the first place, this would safe us the allocation!

GHC exploits such optimisation potential using the worker/wrapper transformation [Gill and Hutton, 2009]. For our purposes, the basic idea is that where we find a strict parameter, we can move all actual work into a dedicated "worker" function. This only leaves the "wrapper" part in the original function definition:

```
eftIntFB = λstep start from to →
  case to of I# to# →
  let wgo = λn# →
        case (>#) n# to# of
          True  → start
          False → step (I# n#) (go (I# ((+#) n# 1)))
      go = λn → case n of I# n# → wgo n#
  in go from
```

Listing 3.20: eftIntFB after Worker-Wrapper Transformation

Now go has become a wrapper function, while wgo implements the associated worker. The trick is that this transformation gives us the opportunity to unfold the wrapper not only when starting the loop on the top level, but also at the recursive call site within the worker function itself:

```
eftIntFB = λstep start from to →
  case to of I# to# →
  case from of I# from# →
  let wgo = λn# →
        case (>#) n# to# of
          True  → start
          False → step (I# n#) (wgo ((+#) n# 1))
  in wgo from#
```

Listing 3.21: eftIntFB with Unfolded Wrapper

After this little dance, we could simply drop the go function, as we are now using the worker function exclusively. In the process we have improved the function significantly: Now the inner loop function wgo actually uses efficient unboxed integers [Peyton Jones and Launchbury, 1991] for tracking the current enumeration value.

However, observe that $n_\#$ still gets boxed when we pass it as a parameter to the step function. The reason for this is that eftIntFB has no information about this function and therefore has to be conservative. In order to optimise further, we need to specialise the code to our step function.

3.4.4 Unfolding

Now that we have optimised the eftIntFB function to our satisfaction, let us return to the fac code from Listing 3.15. We know that in order to produce efficient code, we need to specialise the inner loop with our custom stepper function. To this end, we can now simply unfold the definition of eftIntFB in Listing 3.15, which yields us:

```
fac = λn →
  case n of I# to# →
  let wgo = λn# →
        case (>#) n# to# of
          True  → I# 1#
          False → timesInt (I# n#) (wgo ((+#) n# 1))
  in wgo 1
```

Listing 3.22: Core after Unfolding

The inner loop is starting to take shape. Note that most of our optimisations up to this point have been instances of unfolding function definitions at their call sites, followed by local transformations. In fact, due to the ubiquity of small functions in Haskell code, in-lining transformations are one of the most common optimisations performed on functional programs. Note however how this time around we substantially increased the code size relative to the last version of fac. This should hint at the fact that in practice we need to take care, as excessive unfolding could lead to exponential code size increase [Peyton Jones and Marlow, 2002].

But let us continue with our optimisations. We can now unfold timesInt as well to expose the primitive operator:

```
fac = λn →
  case n of I# to# →
  let wgo = λn# →
        case (>#) n# to# of
          True  → I# 1#
          False → case wgo ((+#) n# 1) of
                    I# r# → I# ((*#) n# r#)
  in wgo 1
```

Listing 3.23: Core after Nested Unfolding

This bring us close to the finish line. However wgo still boxes and unboxes its return value at every step in the recursion. We can correct this using another worker/wrapper transformation, which yields the final code:

```
fac = λn →
  case n of I# to# →
  let wgo = λn# →
        case (>#) n# to# of
          True  → 1#
          False → case wgo ((+#) n# 1) of
                    r# → (*#) n# r#
  in I# (wgo 1)
```

Listing 3.24: Core after Worker/Wrapper

3.4.5 Status Report

At this point we have exhausted every local Core transformation that we could do. The code from Listing 3.24 is exactly what GHC would produce if we used the fac implementation from Listing 2.1. And even though simplified Core is by far not the last step in the compilation pipeline, it is noteworthy that if we assume a basic performance model [Peyton Jones, 1992] this low-level functional code already says a lot about the performance we can expect from the program.

Incidentally, we noted earlier that our fac implementation still had a performance problem. We can now see why, simply by taking a good look at Listing 3.24: The function wgo needs to call itself recursively, and only multiplies the current counter n# with the product once the recursive call returns. Running this program, we would be forced to keep n# in a stack frame until the recursive call returns. At the deepest point this would mean to# nested stack frames, so our maximum stack size would be $O(n)$. As factorial can be computed in constant space, this is clearly not efficient[3].

Thinking about this issue further, we might even recognise that the roots for this behaviour run deep: We are folding from the right using foldr and a strict operator, but actually *generate* the numbers starting from the left. And to make matters even more troublesome, we can not even fix this simply by reversing the direction of the fold as proposed back in Listing 2.3 on page 16. The reason is simply that foldl can not be fused using GHC's short-cut fusion techniques, so all the rules that served us so well would simply stop matching. Resolving this properly would require more principled changes to the fusion framework [Coutts et al., 2007a].

3.4.6 Arity Analysis

However as it turns out this performance problem can be fixed without the need to reinvent fusion. The key insight is that the code in Listing 3.24 is inefficient because it accumulates the counter before the recursive descent, but performs the multiplication for the final result on the way back. As multiplication is associative, it should be possible to move the product calculation in front of the recursion. This would leave no work to be done after the recursive call, so the compiler could apply the tail-call optimisation. This would effectively implement the C loop from Listing 3.1 on page 23.

As it happens, we can reach this goal by changing fac as follows:

```
fac :: Int → Int
fac n = foldr (λx c i → c (i∗x)) id [1..n] 1
```

Listing 3.25: Fixed Haskell Program

For this solution, the programmer has figuratively "leaned so far right [s]he came back left again" [Ruehr, 2001]: We use a *right* fold to construct a function that performs a *left* fold, which we then call with the parameter "1". In the lambda passed to foldr, we

[3]Our implementation will actually go wrong even before becoming inefficient due to overflowing the value range of Int#. As we are only interested in performance, we simply see this as a feature.

have the current counter as x, the function that will continue evaluation for the *right* part of the list as c, and the current product we got from the *left* of the list as i.

This is probably easiest to understand with a small example:

```
foldr (λx c → c . (*x)) id [1..3] 1
= (λc → c . (*1)) ((λc → c . (*2)) ((λc → c . (*3))
    id)) 1
= (λc → c . (*1)) ((λc → c . (*2))
    (id . (*3))) 1
= (λc → c . (*1))
    (id . (*3) . (*2)) 1
= (id . (*3) . (*2) . (*1)) 1
= (((1 * 1) * 2) * 3)
= foldl (*) [1..3] 1
```

Listing 3.26: Fold Emulation

Note how the continuation function c "accumulates" the operations in the order appropriate for a left fold. Once we reach the top level, we apply the function to the start value 1 in order to obtain the factorial.

It might seem like we merely obfuscated our implementation, but we will see that this actually happens to correct the exact problem our first version had. We saw that foldr ended up producing a function that ran left-to-right, and now the order in which calculate the product reflects that! In this example, by way of short-cut fusion and unfolding eftIntFB we obtain the following Core:

```
fac = λn →
  case n of I# to# →
  let wgo = λn# →
        case (>=#) n# to# of
          True  → (λi → i)
          False → let x = I# n#
                      c = wgo ((+#) n# 1)
                  in λi → c (timesInt i x)
  in wgo 1 (I# 1)
```

Listing 3.27: Fixed Haskell Program after Unfolding

It is easy to see that this is simply the same code as in Listing 3.22, except that we replaced the step and next functions, and pass a literal 1 to function returned by wgo.

Now we can make use of a basic feature of functional languages inherited from lambda calculus: A function that takes one parameter and returns a function with another parameter is virtually the same as a function that takes two parameters. We can especially convert the two forms into each other simply by doing an η-expansion ($f = e \quad \Longrightarrow \quad f = \lambda i \to e\ i$). If we do this for the wgo function, float the application inwards and β-reduce where possible, we get:

```
fac = λn →
  case n of I# to# →
  let wgo = λn# i →
        case (>=#) n# to# of
           True  → i
           False → wgo ((+#) n# 1) (timesInt (I# n#) i)
  in wgo 1 (I# 1)
```

Listing 3.28: Fixed Haskell Program after η-Expansion and β-Reduction

Comparing with the program before, we see that we have simply "floated" the lambda from the case branches upwards towards the definition of wgo. Note that as simple as this transformation is, finding the right situations to do this can actually be quite tricky for the compiler: After all, we have to acknowledge the possibility that the program might first pass one parameter to wgo and then proceed to call the partially applied function for many different values of i. The η-expanded code would then re-do the check of n# against to# for every call to the partially applied function!

However, in this case arity analysis decides that the cost for the check is minor relative to the cost of constructing and calling a partially applied function. We can also see that wgo always gets called with two parameters, so this transformation is in fact safe [Breitner, 2004]. Going forward, strictness analysis now tells us that i always gets evaluated by wgo: Either it is the result, or it gets used by the (strict) timesInt function to compute the new i parameter for the recursive call. Therefore we can conclude by induction that wgo itself is strict in that parameter. This is what we need to know for worker/wrapper, so now we are free to displace i in favour of i#:

```
fac = λn →
  case n of I# to# →
  let wgo = λn# i# →
         case (>=#) n# to# of
           True  → i#
           False → wgo ((+#) n# 1) ((*#) i# n#)
  in I# (wgo 1 1)
```

Listing 3.29: Fixed Haskell Program after Worker/Wrapper

At which point we have optimised the code to completion.

3.4.7 Observations

If we compare the result from Listing 3.29 to the Core code we obtained in Listing 3.24 on page 35, we observe that paradoxically introducing our "trick" has actually made the optimised code *shorter*[4]. The new code is also significantly *faster*: The compiler can implement the tail-call simply by updating the parameters followed by a "jump" back to the function entry. Therefore no stack management is needed, and the back-end will compile our function into a tight loop.

Yet we have reached this result by "cheating" a little: We specialised our code to the exact optimisations that we knew the compiler would perform. This not only makes it highly non-obvious why we implemented the function in this way, but also means that the performance of our code is fairly fragile. It would be reasonable to expect that for a different compiler or base library our "trick" might stop working completely. Worse, given that we introduced extra code, our new version might end up slower!

This is why generally speaking, the better approach for writing fast programs is to be explicit about performance-relevant aspects of the program. For example, we could simply write out the loop from Listing 3.29 directly. This might be unsatisfying, as we are essentially writing the Haskell equivalent of the low-level C solution from Listing 3.1 on page 23, but at least we could be reasonably sure about the performance characteristics. Fortunately though, there is a better option here. After all, as we explained the real problem is the direction of the accumulation. So why not simply use `foldl`? As it happens, due to the work of Breitner [2004] this is now actually a good idea, because `foldl` is now defined as follows:

```
foldl :: ∀ a b. (b → a → b) → b → [a] → b
foldl k i0 xs = foldr (λx c i → c (k i x)) id xs i0
```

Listing 3.30: Definition of `foldl`:

Which implements exactly the trick we explained above. Note that this is base library code, and therefore bound to a specific compiler version and therefore optimisation pipeline! Therefore we can safely "import" this behaviour and could implement a fast factorial function as follows:

```
fac' :: Int → Int
fac' n = foldl (*) 1 [1..n]
```

Listing 3.31: Factorial with `foldl`

We will however continue to reference the `foldr` version `fac` throughout this work, as we show how to track down the explained performance problem using profiling.

[4]Which is by no means specific to Haskell – even for low-level languages getting a very specific result can often involve complex preparations.

3.4.8 Case-Of-Case

At this point we have exhausted all optimisations that could possibly apply to our trusty factorial function. However, we have still barely scratched the surface of all the different transformations that play a role when translating a Haskell program into its final form. We could never hope to cover these completely – especially considering that optimisation is a topic that is moving rather quickly, and many transformations only become relevant in relatively specialised domains. However, we will still cover one more rule for its generality and potential to substantially change the control flow of the program: The case-of-case transformation [Peyton Jones and Santos, 1998]. Consider the example from the cited paper:

```
if (not x) then e1 else e2
⟹ case (case x of True → False; False → True) of
       True  → e1
       False → e2
```

Listing 3.32: Control Flow

After desugaring and unfolding the obvious definition of `not`, we are left with two nested `case` expressions, which is rather unsatisfactory. After all, it is clear that the branch taken in the inner `case` will determine the branch taken on the outer level, therefore we should be able to eliminate one. And we can, if we simply copy the outer `case` expression into the branches of the inner:

```
case x of True  → (case False of True  → e1; False → e2)
          False → (case True  of True  → e1; False → e2)
 ⟹ case x of True  → e2
              False → e1
```

Listing 3.33: After Case-Of-Case

This way of simplifying control flow is known for imperative languages as well, where it takes the form of short-cutting labels for the different branches [Aho and Ullman, 1977, section 7.8]. For most languages this represents a special case, as non-strict semantics are restricted to boolean operators. In Haskell however, this technique applies no matter what data structure the `case` expression scrutinises. We can even generalise this to constructors with parameters:

```
case (case f x of C a b → (a, b)) of (a, b) → ...
⟹ case f x of C a b → case (a, b) of (a, b) → ...
⟹ case f x of C a b → ...
```

Listing 3.34: Case-Of-Case for Tuples

Furthermore, if we are smart about let-binding the branches of the outer case expression before the transformation (called "join points" by Peyton Jones and Santos [1998]), we can severely limit the amount of code duplication as well. It is not hard to see why this might be an issue, just consider an inner case expression with more branches:

```
case (case f x of C a b → (a, b)
                  D c d → (c, d))
  of (a, b) → ...
⟹ let joinPt a b = ...
     in case (case f x of C a b → (a, b)
                          D c d → (c, d))
          of (a, b) → joinPt a b
⟹ let joinPt a b = ...
     in case f x of C a b → joinPt a b
                    D c d → joinPt c d
```

Listing 3.35: Case-Of-Case with Join Point

Now the compiler can decide separately whether duplicating the branch code – meaning inlining joinPt – would be acceptable. As a result, the case-of-case optimisation can be applied pretty much anywhere nested case expressions are encountered, which together with let floating means that optimised Core does not often see case scrutinise anything but a literal or a function application. This transformation effectively "flattens" the local control flow of the program fragment.

3.5 Performance Model

After Core optimisations have run their course, the next step is to translate the program into an imperative intermediate low-level language. To recapitulate, this is the stage where the compiler spells out most of the details of mapping the functional program to hardware: Control flow is encoded using explicit code blocks and jumps, and memory layout gets broken down to the point where the code is reasoning about individual memory words. And even though sometimes the compiler can find optimisation opportunities even at this late stage, this is mostly limited to local transformations, such as figuring out a good register allocation [5].

This is relevant to our considerations: We have now reached the point where we can see the performance characteristics of the eventual execution start to shine through. After all, a heap check generated at this point will most likely stay in the program

[5] Note though that the LLVM back-end [Terei and Chakravarty, 2010] can sometimes still improve tight inner loops using low-level assembly tricks [Stewart, 2010].

all the way to the final binary. And while we might not be able to predict the exact cost amount, we know for sure that the check will not be free of charge. This makes the check a potential performance problem. If we remember our nomenclature from Section 2.2.1 on page 9, this makes heap usage a *verb*: A plausible intermediate cause for bad performance.

In this section we are going to build a catalogue of such resource usage types, an abstract performance model for optimised Core. The abstract costs of the model are meant to directly correspond to our profiling verbs. In contrast to the Core performance model used by Santos [1995, chapter 9] or Sansom and Peyton Jones [1997] we will not base this primarily on an abstract machine such as the Spineless Tagless G-Machine [Peyton Jones, 1992]. Instead, in this chapter we will approach the problem from the practical side, deriving our model from the behaviour of real-world Haskell programs as compiled with GHC and executed on realistic hardware. And while we will not quantify the abstract costs just yet, this already allows us to argue that our performance model covers all relevant runtime costs in some form or another. Therefore our performance model exhaustively characterise the performance behavior of a real-world program. Later in Section 5.7 on page 162 we will associate these verbs with actual measurable performance metrics. This will allow us to transfer the causal analysis of the abstract performance model to real-world performance problems.

3.5.1 Core Preparation

The first step of code generation will be to "prepare" the Core code. The purpose of this is not to improve performance: We just normalise some very specific constructs that would have the same meaning for the purpose of code generation anyway. Further reducing the complexity of our language makes it cleaner to talk about performance, as we avoid repeating the same reasoning for different rules.

In GHC, the full Core preparation phase a good number of transformations, with most of them being inconsequential for our purposes. However, we will assume the following normalisations:

1. Arrange for constructors and primitive operations to appear saturated. We can do this because we know at every point how many parameter they should receive. To implement this, we simply η-expand wherever we find an unsaturated application:

   $$\mathsf{C}\ \mathsf{x} \quad \Longrightarrow \quad \lambda \mathsf{y} \rightarrow \mathsf{C}\ \mathsf{x}\ \mathsf{y}$$

   Listing 3.36: Constructor Saturation

   This is always possible and means that we have to deal with partial functional applications less often.

2. To simplify dealing with shadowing, we ensure that each variable name only gets defined exactly once in the program. Variable normalisation just requires renaming all variables to use unique names, which GHC does relatively early in the compilation pipeline:

```
 let x = ... in let x = ... in ...   w1
⟹ let x0 = ... in let x1 = ... in ...
```

Listing 3.37: Variable Normalisation

3. We want A-normal form [Flanagan et al., 1993], where closures are only applied to variables. This can be achieved simply by let-binding all non-variable arguments:

```
 f (...) y
⟹  let x = ... in f x y
```

Listing 3.38: Conversion to A-normal form

The transformation ensures that we can later cleanly divide concerns between determining the parameter values and performing the function call.

4. Finally we transform strict and primitive let bindings into case statements:

```
 let x = ... in ...
⟹ case ... of x → ...
```

Listing 3.39: Convert strict let to case

The reasoning here is that a case statement implies reducing the scrutinee to weak head normal form, which is what we want to do for strict values. Note that primitive values always have to be evaluated strictly, but even for non-primitive values we can often derive strictness properties using a suitable analysis [Clack and Peyton Jones, 1985].

3.5.2 Abstract Evaluation

After preparation the new language subset is quite suitable for talking about operational semantics. Our next aim is to formulate rules that say how much cost we expect each construct to contribute to a program execution. Note that the concrete costs will often depend on how often and in what context a given construct gets evaluated, meaning that in order to be able to truly talk about costs we need to know how the program *evaluates*. Unsurprisingly, there has been substantial prior work on how to reason about program evaluation in a formal way. In the following we opt to follow in the footsteps of Launchbury [1993] and Sestoft [1997], with only minor modifications to introduce and maintain cost terms.

Let us introduce our notation. We will be using symbols to refer to specific term types as follows:

C – **Constructors** will be upper-case letters.

We assume them to be constants coming from the original program. Each constructor has a given arity, corresponding to the number of value arguments.

x – We will notate **variables** as lower-case letters.

They will be used to refer to names in the program as well as to abstract heap locations for evaluation. We will on occasion generate fresh ones in order to refer to new heap cells.

v – **Values** are the result of evaluating expressions, and might appear on the heap.

As we are going to ignore primitives for now, we only have saturated constructors and lambda values:

$$\begin{array}{rcll} v & ::= & C\ x_1\ x_2 ... & \text{(Constructor)} \\ & | & \lambda y.e & \text{(Lambda)} \\ & | & \bot & \text{(Bottom, Hole)} \end{array}$$

Note that we include the $\bot$ term for undefined behaviour as a value. Making it a proper value is a rather unusual choice, most notably Launchbury's original semantics introduce it only retroactively. We will later make more active use of this both as a value as well as an expression term, therefore it makes sense to emphasise its existence up-front.

e – **Expressions** form our program representation:

$$\begin{array}{rcll} e & ::= & v & \text{(Literal)} \\ & | & e\ x & \text{(Application)} \\ & | & x & \text{(Variable)} \\ & | & \texttt{let}\ \{x_1 = e_1,\ x_2 = e_2,\ \ldots\}\ \texttt{in}\ e & \text{(}\textsf{let}\text{ expression)} \\ & | & \texttt{case}\ e\ \texttt{of}\ \{C\ x_1\ x_2 ... \to e_1;\ D\ y_1\ y_2 ... \to e_2;\ \ldots\} & \text{(}\textsf{case}\text{ expression)} \end{array}$$

Apart from the notation, these expression types correspond closely to the ones used by GHC, which we mentioned in Listing 3.6 on page 28. The most notable difference is that we have now separated constructor applications and function applications, and only allow passing variables in both cases. As we have seen in the last section, this is just a question of preparing the Core code suitably.

Γ – Our **heap** will be notated using upper-case Greek letters, which stand for a mapping of variables to either values or expressions:

$$\begin{array}{rcll} \Gamma & ::= & \Gamma, x \mapsto v & \text{(Value)} \\ & | & \Gamma, x \mapsto e & \text{(Thunk)} \\ & | & \{\} & \end{array}$$

Note that as defined here, values are also expressions, therefore saying that expressions can appear on the heap would be enough. However, heap cells that are not in weak head normal form essentially represent deferred (lazy) computations. Such "thunks" will be handled differently from pure values later on, so it appears useful to introduce this distinction right away.

$\mathcal{O}$ – The main goal for this section will be to identify **costs**. Where we can find them, we will notate them using upper-case letters in calligraphic font.

The idea is that these costs will remain rather abstract. After all, as explained in Section 2.2.1 on page 9, they stand for verbs, which must merely be plausible causes for resource consumption. The verb's identity is about "how" something might contribute cost, therefore we will allocate names by the mechanism in question, such as $\mathcal{H}$ for heap allocation.

θ – Finally, we need a way to talk about the entirety of costs produced by the evaluation of an expression or even the whole the program. For reasons that will become apparent later, we will call bags of costs **profiles**, defined as:

$$\theta \quad ::= \quad \theta_1 + \theta_2 \quad | \quad \mathcal{O} \quad | \quad \emptyset$$

Consistent with the definition of a bag, it should not matter in what order costs are added to a profile. Or put in other words: We assume that the + constructor is both associative as well as commutative. We will also make use of the notation $n \times \mathcal{O} = \sum^n \mathcal{O}$.

Pulling all parts together, we can now state what a (cost) **judgement** will look like:

$$\Gamma : e \Downarrow_\theta \Gamma' : v$$

This notation is again heavily inspired by Launchbury [1993]. We state that starting from a certain heap Γ and expression e, evaluation yields a new heap Γ′, a value v as well as a certain amount of costs θ. In the rest of this section we will explain how we can derive such cost judgements for any program state $\Gamma : e$.

3.5.3 Registers and Stack

Before we start with building our abstract model, let us consider what we know about how programs execute. Given the speed at which computational units run in this day and age, a significant portion of run-time cost will simply come from delays involved in obtaining data and saving back results [Patterson, 2004]. The amount of data required for an operation is often a much more reliable predictor of performance than up-front computational complexity. Consequently, we should take special care to account for *storage* costs.

Conceptually, there are three stores in a running Haskell program – registers, the stack and the heap. Let us focus on the first two for now: Registers and stack both hold the current "working" data of the code, such as local variables, parameters or the code pointer. The characteristics of these stores are as follows:

1. Registers are, by construction, the fastest possible memory to access from the CPU. Therefore it is the first and default choice for holding data. Unfortunately, the register file is rather small, and has to be shared across all recursive calls.

2. On the other hand, access to the stack is generally slower, yet it has the advantage that we can grow it to virtually unlimited size. Typically the stack is used to "spill" values that we can not fit into registers.

Allocating register and stack words in a way that optimises performance is a highly non-trivial problem [George and Appel, 1996] that we will not attempt to model here. Instead, for cost we will simply opt to regard registers and stack together as a single compound store. This essentially makes register allocation an advanced caching technique for the top-most stack frame.

The reasoning is that even if a value might end up never touching the stack, every allocated register has an associated "opportunity cost":

- Register pressure might force other register values to get spilt to the stack

- Calls might clobber our register, so we have to save it back

Bottom line is that while it might be quite hard to pin down exactly, every data word put into a register will often become a data word allocated on the stack. Where register pressure becomes a performance problem, it is almost assured to do so via increased stack allocation. Hence we have causation, and therefore can unite both influences together using the "stack allocation" verb, represented by the abstract cost $\mathcal{S}$.

3.5.4 Heap

In contrast to stack data, the life of heap objects is not bound directly to a certain piece of code being executed. Instead, its function is to enable sharing of complex data structures between loosely connected code pieces. As a result, even figuring out whether or not a heap object can be discarded means running a garbage collection heap analysis at program run-time. This pass will have to copy a significant amount of heap data, which means that it will become more complex the more that we allocate but cannot discard [Sansom and Peyton Jones, 1993].

That being said, the allocation of a heap object itself is very cheap [Appel, 1987]. All we need to do is increase the global heap pointer and write the header word and all data into the freshly reserved heap region (see Section 3.5.9). Yet the implied cost due to triggering a garbage collection still means that programs with a high heap allocation quickly suffer in terms of performance. We will therefore assign an abstract cost unit $\mathcal{H}$ for every word that gets allocated on the heap.

In addition to allocation we have to acknowledge that keeping a large heap around is also a significant factor in slowing the program down. For example, even though modern multiple-generation garbage collection will try to not walk the complete heap too often, a large heap will still cause the average time needed for a garbage collection to increase. More subtly, having a large heap makes it less likely for its contents to fit into processor caches, causing general slowdowns due to reduced memory locality. Therefore there is clearly a heap residency verb. However we will not formalise garbage collection here, and will therefore not associate an abstract cost with it.

3.5.5 Constructors

Now we have covered the groundwork, so let us start looking at concrete expression evaluation. We will always do this in a certain way: First we will have a look at how Launchbury-style semantics encode operational meaning, followed by a more thorough exploration of how GHC would implement these semantics for a real Haskell program. Finally we will go back to the rule to insert the appropriate cost term and obtain a rule for our performance model.

First up, we have constructor applications, which require just a simple rule:

$$\Gamma : C\ x_1\ x_2 ... \Downarrow_{\mathcal{O}(C\ x_1\ x_2)} \Gamma : C\ x_1\ x_2 ...$$

Constructors are both expressions and values, and variables uniquely identify their values, therefore we can simply return the expression as our return value.

On the other hand, the cost term $\mathcal{O}(C\ x_1\ x_2)$ is still an unknown for now. To find out what kinds of costs we have to expect, we have to consider this evaluation from

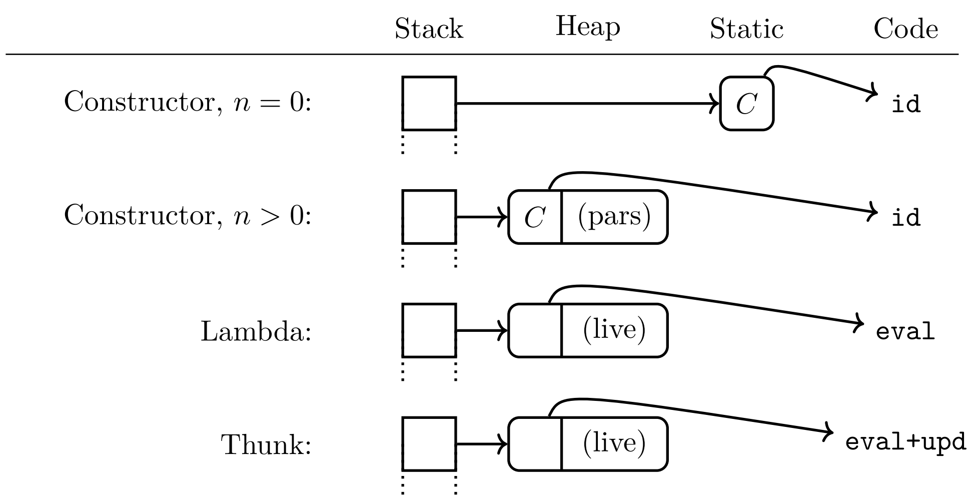


Figure 3.2: Heap Object Construction

the perspective of the compiler. This makes things a bit more involved: Disregarding unboxed tuples or primitives, programs will refer to data structures using pointers, as this reduces the amount of copying the program has to do when passing data around. How we get there depends on the arity of the constructor. Consider for example the nullary constructors of the Bool algebraic data type:

```haskell
data Bool = True | False
```

Listing 3.40: Definition of Bool

As Haskell is not concerned with object identity, there is no need to ever have more than one True or False instance per program. Therefore we simply need to statically provide these closures, reducing the "construction" to simply returning a pointer to the prepared memory location. The opportunity cost for holding a pre-allocated closure in static memory can not be causally connected to the constructor application, and will therefore be ignored[6].

On the other hand, where no such short-cut is available we need to actually allocate heap memory in order to hold our data words. In order to build a valid closure for a constructor with arity n we will have to allocate and copy $n + 1$ data words, as shown in Figure 3.2. Remember from Section 3.5.4 that we use the verb $\mathcal{H}$ for the overhead of allocating and copying a word of heap data. Furthermore, we have some constant costs for the heap check and returning a value, which we will call $\mathcal{C}$.

[6]Note that for types like Int or Char the number of constructors makes storage costs significant enough that GHC in fact switches to "normal" allocation for all "rarely" used 0-arity constructors.

This yields us the abstract cost term:

$$\mathcal{O}(C\ x_1\ x_2...) = \mathcal{C} + n \times \mathcal{H}$$

And even though we are not distinguishing between the 0-arity and the n-arity case here, for our purposes this still gets the main message across: Performance problems can either happen because too many constructor applications are evaluated (effect of $\mathcal{C}$ terms dominate), or because they are allocating too much memory (effect of $\mathcal{H}$ terms dominate). As usual, the actual cost behind these terms are not that interesting, as long as we are reasonably sure that we have not overlooked a way in which constructor applications can become expensive *without* triggering one or both of these scenarios.

3.5.6 Lambdas

Our next literal expression type is the lambda: an anonymous function that we can call via application at a later point in program execution. Due to the design of the abstract semantics, we can use the lambda expression as a value as well:

$$\Gamma : \lambda y.e \Downarrow_{\mathcal{O}(\lambda y.e)} \Gamma : \lambda y.e$$

For a real program, things are a bit trickier. After all, a lambda value must contain all information we need for resuming evaluation of the code from a completely different context. We must especially take care of live variables. Recall that while optimising the "tricky" factorial function in Listing 3.27 on page 37 we had the following code fragment:

```
let x = I# n#
    c = wgo ((+#) n# 1)
 in λi → c (timesInt x i)
```

Listing 3.41: Lambda with Live Variables

There was a good reason we tried to optimise this: The lambda expression has two live non-global variables, c and x, which would both be out of scope once the lambda value actually gets applied to a parameter. To evaluate the body meaningfully, we would need a full closure that saves back both of these values in order to restore them later.

In general, constructing a closure for an expression e with lv(e) live variables therefore requires lv(e) + 1 words of heap space – a code pointer and the values from the context (see Figure 3.2). If we again assign a fresh symbol $\mathcal{L}$ to stand for the constant cost of the lambda's evaluation we can therefore state:

$$\mathcal{O}(\lambda y.e) = \mathcal{L} + \text{lv}(e) \times \mathcal{H}$$

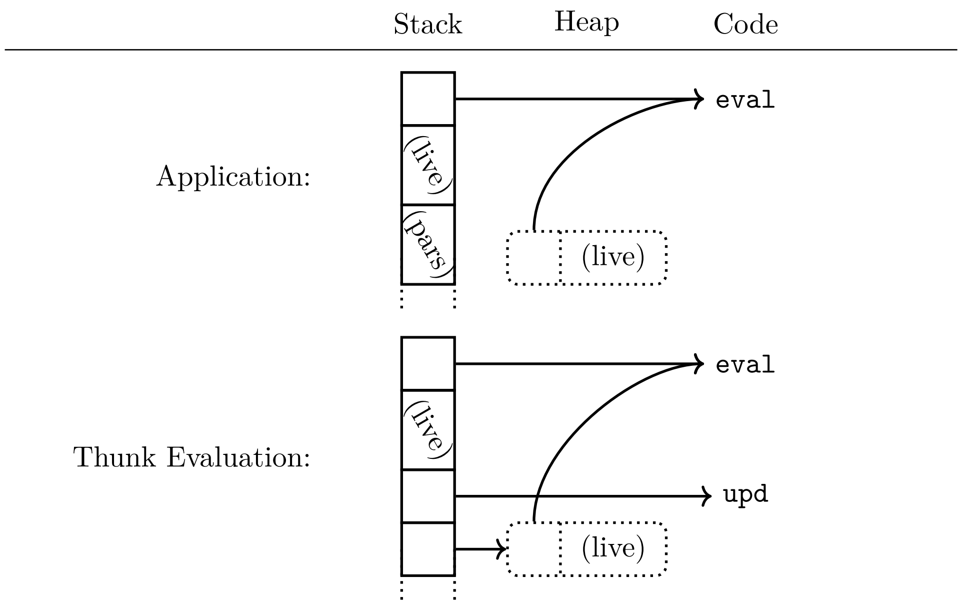


Figure 3.3: Call Implementation

As with constructors, we see two distinct ways in which lambda expressions can cause performance problems. Note however that in contrast to constructor arity, the number of live variables is a performance factor that is quite easy to overlook. For example, the above-mentioned potential problem with the factorial function is not obvious given the original source code from Listing 3.25 on page 36.

3.5.7 Applications

Applications are the counterpart to lambdas, they come into play once the program wants to call the closure with a given set of parameters:

$$\frac{\Gamma : e \ \Downarrow_{\theta_f} \ \Gamma' : \lambda y_1.\lambda y_2....e_b \qquad \Gamma' : e_b[x_i/y_i \cdots] \ \Downarrow_{\theta_b} \ \Gamma'' : v}{\Gamma : e \ x_1 \ x_2 \cdots \ \Downarrow_{\mathcal{O}(e \ x_1 \ x_2 \cdots) + \theta_f + \theta_b} \ \Gamma'' : v}$$

This time we need to take a closer look at the abstract semantics. For an application $e \ x_1 \ x_2 \cdots$ we first need to evaluate e into a closure of matching arity, which in our notation is written as $\lambda y_1.\lambda y_2....e_b$ [7]. We then bind the parameters for the lambda body e_b by substituting the parameter names by the variables from the application (which stand for heap objects). Finally, we can perform the actual call by evaluating the prepared body, passing all evaluation results through.

To summarise runtime costs, we need to account for the two nested evaluations θ_f

[7]Note that in contrast to Launchbury [1993] our application rule binds multiple parameters at the same time. The sole purpose of this is to bring the semantics more in line with actual Haskell execution.

and θ_b plus the expenses for the actual call $\mathcal{O}(e\ x_1\ x_2 \cdots)$. The main work for a call is to re-arrange the context to accommodate the body's evaluation: Parameters and live variables have to be arranged to appear at predictable locations on the stack. Unless we can derive the application to be a tail call, we also want to be able to return to our old code position once the call is finished, so we need to push the new code pointer on top of the stack (refer to Figure 3.3).

As established in Section 3.5.3, we have one unit of abstract cost $\mathcal{S}$ for allocating and initialising a word on the stack. Additionally, we will assign $\mathcal{A}$ to the unique constant costs of an application. Then we have the following expenses for calling a lambda with arity n and m live variables:

Operation	Cost
Pass parameters	$n \times \mathcal{S}$
Restore live variables	$m \times \mathcal{S}$
Push new code address	$\mathcal{S}$
Jump to code, Return	$\mathcal{A}$

Which gives us:

$$\mathcal{O}(e\ x_1\ x_2 \cdots) = \mathcal{A} + (1 + n + m) \times \mathcal{S}$$

So clearly calls can quickly become expensive in terms of stack usage. Remember the optimised Core code of our first version of factorial in Listing 3.24: The recursive `wgo`-calls allocated so much stack space that it caused a performance problem. Even more insidiously, note that the cost can rise both due to the number of parameters passed as well as due to live variables – still a tricky property to control, as explained in the last section. We will explain a cheaper way to implement calls in Section 3.5.11.

3.5.8 Lets & Thunks

The purpose of the `let` expression is to bind a certain sub-expression to a name. This allows its evaluation result to be shared. The abstract formulation looks like follows:

$$\frac{\Gamma, y_i \mapsto e_i[y_i/x_i, ...], ... : e[y_i/x_i, ...] \ \Downarrow_\theta \ \Gamma' : v}{\Gamma : \texttt{let}\ \{x_i = e_i, ...\}\ \texttt{in}\ e \ \Downarrow_{\mathcal{O}(\texttt{let}\,\{x_i=e_i,...\}\,\texttt{in}\ e)+\theta} \ \Gamma' : v}$$

Most of the rule is concerned with replacing original names x_i by y_i, which we assume are fresh for our evaluation. This is required as without renaming the abstract heap Γ might already contain x_i bindings originating from an earlier evaluation of the same `let` expression [Sestoft, 1997]. Apart from that, the purpose of the rule is simply to add the bound expressions e_i to the abstract heap, and proceed to evaluate the body e. The resulting profile θ, abstract heap Γ' as well as the return value v all get passed through as the evaluation result of the `let` expression. However, we still have to determine the `let` expression's cost contribution $\mathcal{O}\,(\texttt{let}\ \{x_i = e_i, ...\}\ \texttt{in}\ e)$.

To determine the cost, we have to first realise that we actually have two cases here: If a bound expression happens to be a constructor or lambda, the value added to the abstract heap is actually already in normal form. Therefore the runtime equivalent is to construct the proper evaluated values right away. As we established in Sections 3.5.5 to 3.5.6 on pages 47–49 this has the abstract costs:

$$\mathcal{O}(x = C\ x_1...) = \mathcal{C} + n \times \mathcal{H}$$
$$\mathcal{O}(x = \lambda y.e) = \mathcal{L} + m \times \mathcal{H}$$

with n the arity of the constructor or m the number of live variables respectively.

For non-literal expressions on the other hand, this is where lazy evaluation comes in: Instead of mapping the variable to a value, the abstract semantics push the full expression e itself. Similarly, a running Haskell program would construct a delayed computation – a *thunk*. For the purpose of construction, this works exactly the same as building a hypothetical 0-arity lambda closure. We allocate a new heap object that carries a special code pointer as well as all live variables that we need to restore to resume its computation. We therefore obtain for a non-literal e with m non-global live variables:

$$\mathcal{O}(x = e) = \mathcal{L} + m \times \mathcal{H}$$

To put things together, we have to finally account for the fact that for each new variable we will need to save a pointer in the local context. We will account for this using the cost $\mathcal{S}$ per binding, which gives us:

$$\mathcal{O}\,(\texttt{let}\ \{x_i = e_i, ...\}\ \texttt{in}\ e) = \sum_i (\mathcal{S} + \mathcal{O}(x_i = e_i))$$

So apart from the relatively minor cost of keeping the pointers around, most work for the `let` expression comes from the number and nature of the heap objects.

3.5.9 Variables

As thunks behave a lot like 0-arity lambdas, we can understand executing variable expressions as performing 0-arity applications. However, we want every thunk to be executed at most once, which we achieve by updating the abstract heap:

$$\Gamma, x \mapsto v : x\ \Downarrow_{\mathcal{O}_v(x)}\ \Gamma, x \mapsto v : v$$

$$\frac{\Gamma : e\ \Downarrow_{\theta}\ \Gamma' : v}{\Gamma, x \mapsto e : x\ \Downarrow_{\mathcal{O}_e(x)+\theta}\ \Gamma', x \mapsto v : v}$$

To summarise, if we find that the heap cell already contains a value, we can simply return it without making any adjustments to the heap. However if the heap contains an expression (a thunk), the rule evaluates it and writes the value back to prevent repeated evaluations.

On the runtime side of things, there is an easy way to implement this rule: We simply have variable evaluation call the heap object's entry code. For a thunk this will evaluate the body and arrange for updating the heap, meaning pushing an extra stack frame of two words and perform the actual update once evaluation returns (see Figure 3.3). On the other hand, values will have a dummy "`id`" entry code that simply returns itself right away, as sketched in Figure 3.2 [8]. Therefore we get:

$$\begin{aligned}\mathcal{O}_v(x) &= \mathcal{V} + \mathcal{S}\\ \mathcal{O}_e(x) &= \mathcal{V} + (3+n) \times \mathcal{S} + \mathcal{U}\end{aligned}$$

with $\mathcal{V}$ standing for variable evaluation and $\mathcal{U}$ for the raw cost of updating a heap cell.

3.5.10 Case

To finish our discussion of operational semantics, we have only `case` expressions left. Their purpose is to extract data from the heap cells built by constructor applications according to Section 3.5.5. The operational semantics are as follows:

$$\frac{\Gamma : e \ \Downarrow_{\theta_2} \ \Gamma' : C_i \ y_j ... \qquad \Gamma' : e_i[y_j/x_j, ...] \ \Downarrow_{\theta_3} \ \Gamma'' : v}{\Gamma_1 : \texttt{case}\ e\ \texttt{of}\ \{C_i\ x_j ... \to e_i, ...\} \ \Downarrow_{\mathcal{O}(\texttt{case}\ ...)+\theta_2+\theta_3} \ \Gamma'' : v}$$

So we first evaluate the scrutinee expression e, then depending on the returned constructor C_i choose the appropriate branch e_i. In the second steps, we bind the constructor fields y_j to the desired variable names x_j using substitution and finally proceed to evaluate e_i. All results get passed through as usual.

Note that again we have a duality with constructors, which is unsurprising from a theoretical point of view. After all Church encoding would see constructors as lambdas and `case` expressions as applications. For actual Haskell programs, we need to evaluate the scrutinee and identify the constructor behind the pointer. Then we can proceed with the appropriate branch, pushing constructor parameters on the stack as required.

Recognising a constructor is relatively straightforward, as we know from Section 3.5.4 that every heap object has a header word. Apart from the code, this header pointer also gives us an *info table* which identifies the constructor type [9]. The cost for choosing

[8] In fact GHC optimises this further by "tagging" pointers according to what they are known to point to [Marlow et al., 2007]. However there is no guarantee that we can always skip evaluation of `id`, so we only discuss this method here.

[9] As with thunks, this can often be detected directly from the pointer tag, saving one indirection

the right branch now depends on the number of branches we have: If we could derive that there is only one possible branch, we have to do no extra work. On the other hand, if there is more than one branch to choose from, we have to either use a branch tree or a jump table to identify the right choice. The compiler will chose whatever seems fastest for the task at hand, with the constant overhead of a jump table being the default solution.

Binding n variables means pushing the appropriate number of stack words, which yields us the sum:

$$\mathcal{O}(\texttt{case}\ e\ \texttt{of}\ \{C_i\ x_j ... \rightarrow e_i, ...\}) = \mathcal{E} + n \times \mathcal{S}$$

where again $\mathcal{E}$ stands for all constant costs for `case` expressions together.

3.5.11 Let-No-Escape

At this point we have covered all expression types of our abstract language. However, there is still one interesting case left to cover: Remember that one of the more expensive and unpredictable parts of constructing and calling a function closure was saving and restoring the live variables. However, this is not always necessary. After all, the live variables might still be in scope when the function body gets executed. Suppose our code was shaped as follows:

$$\begin{array}{l}\texttt{let}\,\{f = \lambda x. ...; ...\}\\ \ \texttt{in case}\ ...\ \texttt{of}\,\{C \rightarrow f\ ...; D \rightarrow f\ ...\}\end{array}$$

Assuming that these are the only two references to f, we know that we will have exactly the same live variables at the application site as we had when the `let` expression was executed. This means that there is actually no need to either save or restore live variables: f can simply use the data that is already on the stack!

For GHC, there is a special optimisation pass at the end of Core generation that detects such “let-no-escape” optimisation opportunities. The critical part is to prove that no closure reference “escapes”, meaning that all calls to the function are contained in the `let` body. This implies that the function application has strictly fewer live variables than the application site, therefore making restoring live variables redundant. In the end, this means not only that the function application can prepare the stack rather quickly, but it can also short-cut the jump to the known function entry code.

This means that “let-no-escape” function applications actually involve no heap access whatsoever, making them substantially faster than standard calls. In fact, this optimisation is of tremendous importance for generating fast programs. To see why,

$$\Gamma : C\ x_1\ x_2 ...\ \Downarrow_{\mathcal{C}+n\times\mathcal{H}}\ \Gamma : C\ x_1\ x_2 ... \tag{Con}$$

$$\Gamma : \lambda y.e\ \Downarrow_{\mathcal{L}+\mathrm{lv}(e)\times\mathcal{H}}\ \Gamma : \lambda y.e \tag{Lam}$$

$$\frac{\Gamma : e\ \Downarrow_{\theta_f}\ \Gamma' : \lambda y_1.\lambda y_2 ....e_b \qquad \Gamma' : e_b[x_i/y_i \cdots]\ \Downarrow_{\theta_b}\ \Gamma'' : v}{\Gamma : e\ x_1\ x_2 \cdots\ \Downarrow_{\mathcal{A}+(1+n+\mathrm{lv}(e))\times\mathcal{S}+\theta_f+\theta_b}\ \Gamma'' : v} \tag{App}$$

$$\frac{\Gamma, y_i \mapsto e_i[y_i/x_i, ...], ... : e[y_i/x_i, ...]\ \Downarrow_{\theta}\ \Gamma' : v}{\Gamma : \texttt{let}\ \{x_i = e_i, ...\}\ \texttt{in}\ e\ \Downarrow_{\sum(\mathcal{S}+\mathcal{O}(x_i=e_i))+\theta}\ \Gamma' : v} \tag{Let}$$

$$\Gamma, x \mapsto v : x\ \Downarrow_{\mathcal{V}+\mathcal{S}}\ \Gamma, x \mapsto v : v \tag{Var1}$$

$$\frac{\Gamma : e\ \Downarrow_{\theta}\ \Gamma' : v}{\Gamma, x \mapsto e : x\ \Downarrow_{\mathcal{V}+(3+\mathrm{lv}(e))\times\mathcal{S}+\mathcal{U}+\theta}\ \Gamma', x \mapsto v : v} \tag{Var2}$$

$$\frac{\Gamma : e\ \Downarrow_{\theta_2}\ \Gamma' : C_i\ y_j ... \qquad \Gamma' : e_i[y_j/x_j, ...]\ \Downarrow_{\theta_3}\ \Gamma'' : v}{\Gamma_1 : \texttt{case}\ e\ \texttt{of}\ \{C_i\ x_j ... \to e_i, ...\}\ \Downarrow_{\mathcal{E}+n_i\times\mathcal{S}+\theta_2+\theta_3}\ \Gamma'' : v} \tag{Case}$$

Figure 3.4: Performance Model

take another look at the above code: All we do is evaluate the body of f for two different values of x. This type of control flow appears quite often in functional programs: Quite a bit of the ultimate performance of our factorial implementation was based on the fact that the let-no-escape optimisation would apply to our `wgo` function from Listing 3.23 on page 35 forward. Furthermore, the case-of-case transformation we introduced in Section 3.4.8 on page 40 will deliberately generate this type of code for join points.

For our purposes, this rises the question: Do we need to modify our performance model in order to account for the let-no-escape transformation? Fortunately, this transformation has virtually no overhead. This means that for the sake of argument, we could only ever make *absence* of this transformation a verb. This however makes no real sense for causal reasoning, which is why we can simply ignore this case for our performance model.

3.5.12 Conclusion

This concludes our abstract performance considerations. Figure 3.4 shows a summary of the rules we derived. This allows us to reason about real Haskell programs: Suppose we take a Haskell program and evaluate the equivalent of its Core code with these rules. Then we know that any possible performance-related effect of the program run would be reflected in our abstract model by one or more abstract cost entities. This is exactly what we want, as the next chapter will show how we can trace causality for these back to their respective root causes.

Chapter 4

Causality Analysis

"Subjected to the influence of so many gods, who both protect and threaten them, the nations of antiquity would like to have advance notice of their whims."

— *Asterix and the Soothsayer, René Goscinny & Albert Uderzo*

The past chapter established the main poles of the profiling problem: The first one is represented by all the possible causes of performance problems. For our purposes this is the entirety of design decisions that went into development of a program, mostly encoded as Haskell source code. On the opposing side, we have all the effects that we want to track. As we have seen, our performance model already enumerates a good number of distinct ways in which the program in question could end up being inefficient. The main question for this chapter is going to be how well we can establish causality between these two sides.

To do that we will systematically employ Lewis' theory of causation [Lewis, 1973] as introduced in Section 2.3 on page 13. As causality judgements critically depend on our choice of closest world, this will not be a completely passive analysis. In fact, we have to be very careful in how we make these choices, as it is quite easy to either over- or under-estimate causality. Note that these two trappings are very different in nature: As we explained back in Section 2.3.5, we expect our analysis to overestimate causality all the time. The reason is that we will never be able to tell essential program components from interchangeable ones. For example, there is simply no way that our analysis could know whether usage of the multiplication operator was in fact required in order to compute a factorial. Hence for a complete profile we will always have to include it as a potential suspect.

On the other hand, underestimating causality would be a severe problem. The reason lies in our methodology: In order to break down the task, we will reason about the causes of intermediate events, and use transitivity to propagate them. This approach relies on the fact that the causes for these intermediate events will get carried forward correctly. If we miss an effect of one of these event, we might end up losing track of arbitrarily many causal connections! Therefore our primary motivation throughout will be to make it easy for us to track down every single effect – even if it means that we are clearly overestimating causality.

This chapter has two parts: We will start by considering causal connections in the absence of program transformations. To do that, Section 4.2 will translate the performance model from the last chapter, and show how we can see it as generating a network of events. This event interpretation will allow us to analyse causal dependencies. Section 4.3 and Section 4.5 on page 80 will show how we can systematically map these causal dependencies to term annotations. Furthermore, Section 3.5.4 on page 47 and Section 4.6 on page 85 will explain how we need to change our viewpoint in order to make causal reasoning about laziness and bindings intuitive. Section 4.7 will then present the finished causality model and explain its properties and intuition.

In Section 4.8 on page 100 we will then formally reason about how code transformations rewrite the causal network given by these semantics. We will see that depending on the type of transformation the effects on the profile can be anything from harmless to fairly severe. However, we will also show that even in the worst case we can still limit the damage to – occasionally rather extreme – cases of causality overestimation, promising that we will never actually miss a cause that exists in the program run.

4.1 Introduction

The performance model we derived in the last chapter tells us what *abstract* costs we would expect if we were to execute a given Core program. The analysis of this will yield us causal connections, which as explained in Section 2.2 on page 8 is one corner stone for abductive reasoning. Later in Chapter 5 we add estimates about *actual* execution cost into the mix, so the user can properly gauge plausibility. For the moment however, we will focus entirely on *causality*. Exactly which parts of the program played a role in shaping a certain part of the profile? Which parts did not actually matter for performance and can therefore be ignored?

We will answer these questions using systematic analysis of the rules formulated in the last chapter. Again we have to account for the dynamic nature of program evaluation: The causal context of a cost will critically depend on the concrete run-time control flow. Building the desired cause terms therefore again involves following program execution. Our approach will be the same as in the last section: By extending Launchbury's operational semantics, we will generate a causal description as a "side-product".

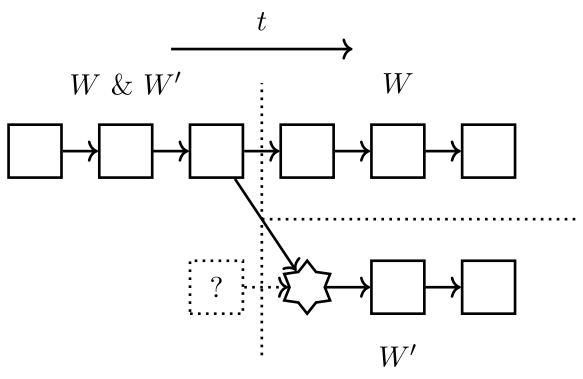


Figure 4.1: Causality Reasoning: Past and Future

4.2 Events

Our goal is to ground our semantics in Lewis-style causality theory [Lewis, 1973] as introduced in Section 2.3. To do that, we have to think about things in the same way we would when trying to make sense of a car crash: What were the *events* that led up to the situation in question, and how are they connected? Which ones were actually essential, and which could be removed without actually making a difference?

When we formulated our cost semantics in Figure 3.4, we used nested rule matches to abstractly describe evaluation. To make it more intuitive to think about this in terms of causality, it is a good idea to think of this in terms of *events*[1]. Consider for example the constructor application rule:

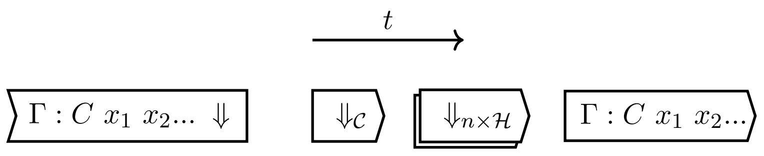
$$\Gamma : C\ x_1\ x_2 ... \Downarrow_{\mathcal{C}+n\times\mathcal{H}} \Gamma : C\ x_1\ x_2 ...$$

We can deconstruct the process of applying this rule using three event types:

Event	Description
$\boxed{\Gamma : e \Downarrow}$	A rule matches input heap pattern Γ and expression pattern e
$\boxed{\Downarrow_{\mathcal{O}}}$	A rule emits a unit of cost $\mathcal{O}$
$\boxed{\Gamma' : v}$	A rule returns a new heap Γ' and a value v.

Clearly we could deconstruct a whole abstract program execution using these events and start reasoning about causation between them. However, we need to be careful. After all, let us think exactly about what we are planning to do: For causal reasoning we expect to depart the original world W by introducing a single "miracle event", which brings us into a parallel universe W' where we can proceed to apply our rules in order to

[1]Our approach here is inspired by the work by Taylor [1993] on causality semantics.



t
$\Gamma : C\ x_1\ x_2 ... \Downarrow$
$\Downarrow_{\mathcal{C}}$
$\Downarrow_{n \times \mathcal{H}}$
$\Gamma : C\ x_1\ x_2 ...$

Figure 4.2: Event Decomposition of a Constructor Rule Match

derive the new "future". For example, back when we introduced causality in Section 2.3 on page 13 this miracle event was a tree ceasing to exist, which led us to a situation where the car crash existed in the original world W, but not in the alternate world W'. Therefore, we reasoned that these two events had a causal relationship.

However, what if we make our "miracle" that the car crash does not happen? Clearly, the car passing through means that there cannot be a tree in W'. So can we conclude that the car crash is what causes the tree to exist? This type of reasoning is clearly dangerous, as it can easily lead us to confusing causes with effects. To ensure consistency of causal reasoning we therefore demand that our events have a temporal order [Lewis, 1979], which tells us in what direction we are "allowed" to look for causation. Figure 4.1 shows how we imagine this to work: Our worlds W and W' are only allowed to differ in terms of events that come *after* the miracle event. This ensures that "past" events can never become effects in our causality analysis.

Fortunately finding a suitable temporal ordering is rather straight-forward for events arising from our operational semantics. After all, our underlying machine execution model is sequential by nature. Therefore we would naturally assume that the left-hand side match happens "before" any potential nested rule matches, the events of which in turn happen "before" the return value gets produced. For example, evaluating a constructor application would give us the event order shown in Figure 4.2: The rule match event "$\Gamma : C\ x_1\ x_2 ... \Downarrow$" followed by a number of subsequent "result" events.

4.2.1 Event Causes

Our goal is now to take the event network of a rule match and run causality analysis on it. Our starting assumption will be that a judgement gets attempted, which will generally cause a rule match, which in turn will give rise to events according to our event decomposition of the rule. Note that the judgement getting attempted is not self-evident: We might be looking at the top-most rule activation of some program evaluation, or we might actually be in the middle of a recursive rule match. In either case we will simply see the judgement getting attempted as axiomatic. We will especially assume that such judgements adhere to the form introduced in Section 3.5.2 on page 43 [2].

[2]Formally speaking, we declare worlds with other judgement/event forms as unlawful[Taylor, 1993].

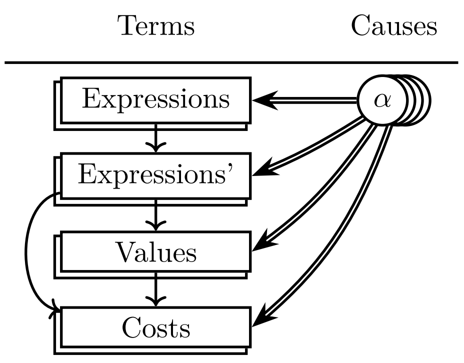


Figure 4.3: Maintaining Cause Links

So our basic scenario is an attempt to obtain a judgement of the form $\Gamma : e \Downarrow_\theta \Gamma' : v$. The concrete rule match will now directly depend on what kind of expression type e is. The cause for this is not immediately known to us, as the answer depends on the history of the passed expression. For example, if our program was simply the desugaring of:

```
let x1 = ...; x2 = ...; in C x1 x2
```

Listing 4.1: Constructor Source

we would say that the source code of the let expression body is what caused us to encounter a constructor here.

On the other hand, we will eventually have to deal with code transformations, which will make the story more complicated. It is important that we address this particular problem generally, as we clearly do not want the complexity of transformations to eventually bleed into our treatment of, say, the constructor application rule. It should make no difference to our causality semantics whether code transformations touched the expressions or not. Instead, our causality semantics should be able to see the causal history simply as a term annotation. Then we can clearly describe what causes a constructor rule match: The fact that a judgement was attempted together with the cause term that was annotated on the constructor expression.

In fact, we will apply this idea generally, as shown in Figure 4.3: Using knowledge about a rule's inner workings, we can derive new annotations for the produced terms using causal transitivity as explained in Section 2.3.5 on page 18. In fact, we will represent the whole causal process using cause-annotated terms: Code transformations will work entirely on cause-annotated expressions, which evaluation consumes in order to produce similarly annotated value and cost terms. In the end, we will have costs annotated with their causal history, which is exactly what we want for performance analysis.

4.2.2 Cause Annotations

An extension to the notation from Section 3.5.2 is in order. The new protagonist for this section will be the *cause term*, corresponding to the profiling-related reasons that a certain event happens for our term. In the end, we should be able to look at the cost term annotations of the top-most judgement and obtain a break-down of the reasons for the program's resource usage.

α – We will use lower-case letters from the beginning of the Greek alphabet to refer to cause terms. We will leave their definition abstract, but think of them as standing for properties or parts of the original program source code as explained in Section 2.3.2.

However we assume that cause terms can be composed using the logical "and" operator $\wedge$. We will make use of a short-hand where it is unambiguous:

$$\alpha \wedge \beta = \alpha\beta$$

As usual the order in which causes are combined does not matter, meaning the $\wedge$ operator should be both commutative and associative.

Cause terms are used to *annotate* the previously defined term types. To be concrete, we will have the following types of annotated terms:

$$\begin{aligned} {}_{\langle\alpha\rangle}e &= \underline{e} && \text{– cause-annotated expression} \\ {}_{\langle\alpha\rangle}v &= \underline{v} && \text{– cause-annotated value} \\ {}_{\langle\alpha\rangle}\mathcal{O} &= \underline{\mathcal{O}} && \text{– cause-annotated cost} \end{aligned}$$

with an underlined symbol such as $\underline{e}$ standing for both the term as well as its annotation. In either case, the annotated term refers to a cause for the expression, value or cost having its present form in context of the event we want to associate it with.

So for example assume that the constructor rule match events shown in Figure 4.2 were the result of matching on the annotated expression term ${}_{\langle\alpha\rangle}C\ x_1\ x_2 \ldots$. Then α stands for the cause of the expression actually being $C\ x_1\ x_2 \ldots$. The initial match event depends causally on this fact, because if we counter-factually assume a different value we would presumably get a different match event, Put generally, let us say we have an event E that successfully matches an annotated term variable such as ${}_{\langle\alpha\rangle}e$. Then we should have:

$$\neg\alpha \;\Box\!\!\rightarrow \neg E$$

Which, again, means that in an alternate world W' with α false we have to assume that e changes and therefore E does not happen.

4.2.3 Annotated Judgements

Changing our term definitions to include annotations means that we have to adjust evaluation as well. For starters, heaps Γ now store *annotated* expressions and values, and profiles θ become more truthful to their name by saving not only the costs, but their causes as well:

$$\begin{array}{rcl} \Gamma & ::= & \Gamma, x \mapsto \underline{v} \\ & \mid & \Gamma, x \mapsto \underline{e} \\ & \mid & \{\} \\ \theta & ::= & \theta_1 + \theta_2 \quad \mid \quad \underline{\mathcal{O}} \quad \mid \quad \emptyset \end{array}$$

If we assemble all these changes together, we now get the new judgement form:

$$\Gamma : \underline{e} \Downarrow_\theta \Gamma' : \underline{v}$$

As explained in Section 4.2.1 we outlaw judgements that do not adhere to this form, so every single "moving part" of the judgement now carries an annotation.

Similarly, we will not explicitly track the cause for the judgement request, even though we could theoretically express it using another annotation:

$${}_{\langle\delta\rangle}\left[\Gamma : \underline{e} \Downarrow_\theta \Gamma' : \underline{v}\right]$$

However there is no need for that. We know that the context term will always tell us that we are either in the top-level judgement or within some nested rule matches. In either case, the entity requesting the judgement has more information about the context than we have. Therefore every rule will just regard the match getting attempted as a precondition and rely on the caller to keep this is mind when interpreting and propagating cause annotations. For example, if a rule requests a judgement conditional on a β, it will have to remember to annotate β on everything that depends on the result of this nested rule match.

Implementing such "context changes" will regularly involve putting extra annotations on existing terms. To keep rules compact, we will use the following short-hands for adding new annotations to already-annotated terms:

$$\begin{aligned} {}_{\langle\alpha\rangle}(\theta_1 + \theta_2) &= {}_{\langle\alpha\rangle}\theta_1 + {}_{\langle\alpha\rangle}\theta_2 \\ {}_{\langle\alpha\rangle}\underline{\mathcal{O}} &= {}_{\langle\alpha\rangle}{}_{\langle\beta\rangle}\mathcal{O} = {}_{\langle\alpha\beta\rangle}\mathcal{O} \\ {}_{\langle\alpha\rangle}\emptyset &= \emptyset \end{aligned}$$

as well as equivalently ${}_{\langle\alpha\rangle}{}_{\langle\beta\rangle}e = {}_{\langle\alpha\beta\rangle}e$ and ${}_{\langle\alpha\rangle}{}_{\langle\beta\rangle}v = {}_{\langle\alpha\beta\rangle}v$.

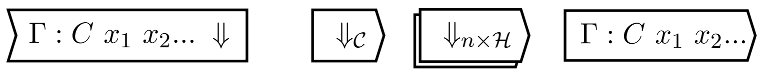

$\Gamma : C\ x_1\ x_2 ...\ \Downarrow$
$\Downarrow_{\mathcal{C}}$
$\Downarrow_{n \times \mathcal{H}}$
$\Gamma : C\ x_1\ x_2 ...$

Figure 4.4: Event Decomposition of a Constructor Rule Match (Repeated)

4.3 Deriving Annotations

Let us return to the constructor application rule. In Figure 4.4 we have repeated the events as they happened in the original world W: We found a constructor application, and returned a bunch of costs as well as a value. So the causal process we are looking at has one "input" event that gets triggered from outside – the match – and a number of "output" events that might in turn trigger other events. Note that we even draw them differently depending on their role: Input events will have a shape that suggests an "incoming" arrow, while the shape of output events shows a matching "outgoing" arrow. This should make it clear that we think of them as two sides of the same coin: The output events of one rule will most likely become the input events of another.

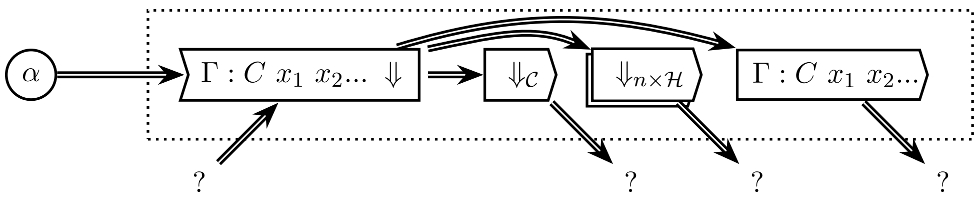
For causal reasoning this distinction decides the annotation responsibility: The terms associated with input events are assumed to already carry valid annotations, while we have to do a causality analysis on the rule in order to decide how to annotate the terms of the output events. To get us started, we know by assumption that we always have some sort of input event of the form "$\Gamma : \underline{e}\ \Downarrow$ ". For a constructor application, we would specifically handle the case that $\underline{e} = {}_{\langle\alpha\rangle}C\ x_1\ x_2...$, which means that α is the cause for the concrete input event shown on the left in Figure 4.4. This is enough to get us started with unravelling the causal processes "inside" the rule application. We can now ask: Which of the other events would change or disappear in a world W' where the initial match event did not happen? To answer this, we are not allowed to assume that the initial match event simply does not happen in W', so the only way to change the course of events is to change e so that it does *not* match $C\ x_1\ x_2...$.

Let us use the recipe from Section 2.3.4 on page 17 and simply use the hole term $\perp$ as a drop-in replacement. Our miracle is therefore that instead of the match event "$\Gamma : C\ x_1\ x_2...\ \Downarrow$ " we somehow find ourselves with the pseudo-event "$\Gamma : \perp\ \Downarrow$ "[3]. As no rule is allowed to match this term, the W' judgement would effectively look as follows:

$$\Gamma : \perp\ \Downarrow_{\perp}\ \Gamma : \perp$$

Note that while strictly speaking we still have a match event, producing $\perp$ is effectively equivalent to no event happening at all. After all, no other rule is allowed to consume

[3]We assume that no matter what situation, we will always be able to name a current heap.



α
$\Gamma : C\ x_1\ x_2 ...\ \Downarrow$
$\Downarrow_{\mathcal{C}}$
$\Downarrow_{n\times\mathcal{H}}$
$\Gamma : C\ x_1\ x_2 ...$
?
?
?
?

Figure 4.5: Causality Relations for a Constructor Rule Match

it, so no further events will get triggered according to this viewpoint. In the real world, our program would probably either freeze or crash outright. In the end, we can say with confidence that a failed match in the alternate world W' would mean that all result events vanish as well.

This means that we have demonstrated a causal connection! Rather unsurprisingly, emitting results does indeed depend on the rule match. Figure 4.5 summarises the causality network as we now see it: In context of our judgement the match is caused by the parameterised cause α. From there the match causes all return term events to happen. We can only assume that these events will then in turn spawn other events in other parts of the program evaluation. For example, the fact that we returned a certain constructor value might well become the reason we take a certain branch later when evaluating a `case` expression.

Consequently, we want to put an annotation on the returned terms to record what we derived about their causal history. For our simple example, it is clear that all three terms depend on α by transitivity over the match event. Formulated as an annotated evaluation rule, this yields us:

$$\Gamma : {}_{\langle\alpha\rangle}C\ x_1\ x_2 ...\ \Downarrow_{\langle\alpha\rangle\mathcal{C}+n\times{}_{\langle\alpha\rangle}\mathcal{H}}\ \Gamma : {}_{\langle\alpha\rangle}C\ x_1\ x_2 ...$$

4.3.1 Variables

Studying the constructor rule should have given us a general idea of how causality analysis works. Let us now consider the slightly more complicated variable evaluation rule from Section 3.5.9 on page 52, for the case that we need to evaluate a thunk:

$$\frac{\Gamma : e\ \boxed{\Downarrow_\theta}\ \boxed{\Gamma' : v}}{\boxed{\Gamma, x \mapsto e : x}\ \Downarrow_{\mathcal{V}+(3+\mathrm{lv}(e))\times\mathcal{S}+\mathcal{U}+\theta}\ \Gamma', x \mapsto v : v}$$

So when evaluating a variable expression that refers to a heap slot that is not in weak head normal form, we evaluate the associated expression and update the heap slot before returning the result value.

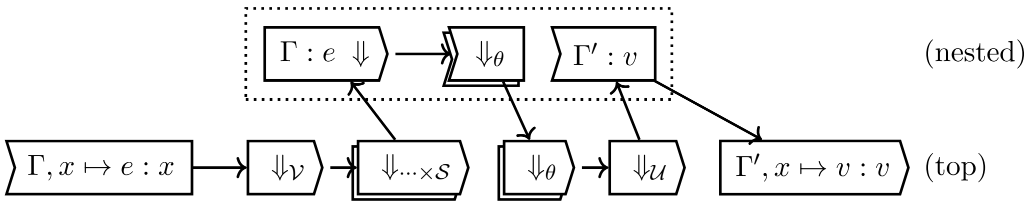


Figure 4.6: Ordered Event Decomposition of a Variable Rule Match

As shown in Figure 4.6 we can decompose a match of this rule into 9 distinct events. Like before the match is an input event triggered from outside, and we have a number of output events corresponding to emitted costs as well as the returned terms. However this time around we actually have a *nested* rule match due to the need to evaluate the expression e. This nesting flips around the responsibilities from our point of view: Initiating the nested rule match becomes an output event, and all costs and results turn into inputs to the rule's causal process. As depicted in Figure 4.6, we end up with 3 input events and 6 output events.

4.3.2 Local Miracles

We already know from Section 4.3 how to handle a mismatch on the initial match event. But this time around this is not our only input event, as the nested rule match returns costs θ and a result value v back to us. What is more, in this case the rule makes no actual "control flow" decisions based on those values. All we do is pass these terms through, so does this simplify our treatment?

Let us again do an alternate world thought experiment. Consider a scenario W' where instead of the full costs θ we end up generating a subset θ' with exactly one cost replaced by $\perp$. In an alternate world implied by this change, we can then judge:

$$\frac{\Gamma : e \Downarrow_{\theta'} \Gamma' : v}{\Gamma, x \mapsto e : x \Downarrow_{\mathcal{V}+(3+\mathrm{lv}(e))\times\mathcal{S}+\mathcal{U}+\theta'} \Gamma', x \mapsto v : v}$$

The difference might be hard to spot without the highlights, and for good reason: The variable rule makes no actual assumptions about the contents of the profile, therefore it applies just as it did in the original world W. The only difference is that now we pass through θ' instead of θ! As it turns out, this time around removing cost has little actual impact on the program evaluation process. We conclude that for every cost event that we receive from the nested judgement we are going to re-emit exactly an equivalent cost event, but nothing else.

It is not hard to see that the same applies to the value v, both in its function as a return value and as the new term to be associated with x on the heap. Just suppose a world W'' where we get a $\perp$ result. Again the rule will match as before, resulting in a $\Gamma', x \mapsto \perp : \perp$ right-hand side. Therefore the result value causes exactly the heap cell and the result value to change.

Consider what this means for annotations: We know that the term annotations names the cause for the exact same terms to re-appear in the output events. Therefore these terms need to carry the same annotations (at minimum). Or put another way: Where the term does not matter, we can simply copy the annotation along with it.

4.3.3 Nested Annotations

If we look at the variable rule again, we see that costs and the value are not the only term that gets passed through. After all, at the start of the rule match we extract the expression e from the heap and pass it through to the nested rule match. As we know, expressions are actually defined recursively, and therefore can contain further sub-expressions. It is quite likely that different parts of the expression might not originate from exactly the same source code, and we could therefore like to associate different cause terms with them. We have just learnt that carrying around such extra annotations does not have to mean extra effort on our side – so can we introduce additional ones here without introducing too much additional complexity?

When we defined annotations in Section 4.2.3 on page 63 we were still working on the back of the original expression definition from back in Section 3.5.2 on page 43. For nested annotations we need to revise this a bit further. We will now use the following syntax to refer to (deeply) annotated values and expressions respectively:

$$
\begin{array}{rcl}
\underline{v} & ::= & {}_{\langle\alpha\rangle}C \ {}_{\langle\beta_1\rangle}x_1 \ {}_{\langle\beta_2\rangle}x_2 ... \\
& | & {}_{\langle\alpha\rangle}\lambda y.\underline{e} \\
& | & \perp \\
\\
\underline{e} & ::= & \underline{v} \\
& | & {}_{\langle\alpha\rangle}e \ {}_{\langle\beta\rangle}x \\
& | & {}_{\langle\alpha\rangle}x \\
& | & {}_{\langle\alpha\rangle}\texttt{let}\ \{x_1 = \underline{e}_1,\ x_2 = \underline{e}_2,\ \ldots\}\ \texttt{in}\ \underline{e} \\
& | & {}_{\langle\alpha\rangle}\texttt{case}\ \underline{e}\ \texttt{of}\ \{C\ x_1\ x_2 ... \to \underline{e}_1;\ D\ y_1\ y_2 ... \to \underline{e}_2;\ \ldots\}
\end{array}
$$

As before we see $\underline{v}$ and $\underline{e}$ as short-hands for annotated terms, which can now contain entire trees of cause-annotated sub-terms. Especially note that we have put annotations on variables for both constructor as well as the function applications. This will come in useful later when we consider `let` expressions in Section 4.6.2 on page 88.

4.3.4 Nested Events

The nested annotations represent something fundamentally new for our event model. After all, introducing the events underlying our semantics in Section 4.2 on page 59 we said that matching a whole expression e – presumably including all its sub-expressions – only took one event. Pulling out just the part of the causal network where the initial rule match of the variable rule causes thunk evaluation to start, we would have thought of it as follows:

$$\begin{tikzcd}
\alpha \arrow[r, Rightarrow] & \boxed{\Gamma, x \mapsto e : x} \arrow[r, Rightarrow] & \boxed{\Gamma : e \Downarrow}
\end{tikzcd}$$

Figure 4.7: Flat Events

Yet the whole point of annotations is that they tell us about the causes of underlying events. Therefore in adding in new annotations we also need new events to anchor them to, making the full causal story a good deal more complicated. For example instead of having only one initial match event as pictured above, we now have a decomposition where the "main event" of the rule match actually is accompanied by a number of "nested events" for all involved sub-terms. So suppose that the initial match was on the term "$\Gamma, x \mapsto {}_{\langle\beta\rangle}e : {}_{\langle\alpha\rangle}x$", then we would have the following event network:

$$\begin{tikzcd}
\beta \arrow[r, Rightarrow] & \boxed{e} \arrow[r, Rightarrow, "?" description] & \boxed{e} \\
\alpha \arrow[r, Rightarrow] & \boxed{\Gamma, x \mapsto \bullet : x} \arrow[u] \arrow[r, Rightarrow] & \boxed{\Gamma : \bullet \Downarrow} \arrow[u]
\end{tikzcd}$$

Figure 4.8: Nested Events

The new "e" event on the left represents the (sub-)event that the initial match happened with that exact value of e. The cause for this is β as indicated by the annotation. Note that these new events are a bit different than the events we considered so far. For example, we want such nesting events to only ever exist in the context of a "main" event explaining where the value in question actually comes into play. We would furthermore expect causal dependencies on them to always imply causal involvement of the "parent" event.

To build some intuition let us briefly consider a real-world parallel: We can view the original match $\Gamma, x \mapsto e : x$ as being an "envelope" for the contained expression e. Just as a filled envelope is a letter without regard for its concrete contents, our match events happens regardless of the actual value of e. This means that it clearly makes sense to consider the causes for container and contents separately. After all, receiving an envelope might have a lot to do with the workings of the postal service, whereas the contents might be the result of an involved word smithing process at a far-away location. On the other hand, we can not fully separate the two processes either. At minimum we would expect that by receiving the envelope, we would automatically come into possession of the contents as well. However this distinction is still useful, as depending on the causal process in question it can depend on either or both parts of the decomposed event.

In our case, the downstream event is "$\Gamma : e \Downarrow$ ". This event takes over the nested annotated expression, so we accompanied it with an equivalent "e" sub-event as well in Figure 4.8. In real life, this new event might be taking the letter in our hands in order to read it, which can still be decomposed into the distinct events of "holding something in your hands" and "that 'something' is a letter". We would probably expect the letter in our hand to depend exactly on what was found in the envelope. We can check this using another closest-world thought experiment: What would happen if e was a $\perp$ term, or our "letter" was simply empty?

$$\begin{tikzcd}
\alpha \arrow[r, Rightarrow] & \boxed{\Gamma, x \mapsto \perp : x} \arrow[r, Rightarrow] & \boxed{\Gamma : \perp \Downarrow}
\end{tikzcd}$$

Figure 4.9: Alternate World

The answer is that in the alternate world we would still take it in our hand, because beforehand we could not know that it was empty. Similarly, the rule would still match and simply pass on the hole term. We conclude by causality transitivity that if we decompose the output event into its components, the event corresponding to the expression term is also caused by the annotation of e, which is β as shown in Figure 4.8.

But let us not forget about the "nesting" part: Just as we decomposed the match event by pulling out the annotated expression, we also ought to separate out all nested sub-expressions contained within e. For example let us suppose that the expression was a constructor application. Then we would have the following full causality graph:

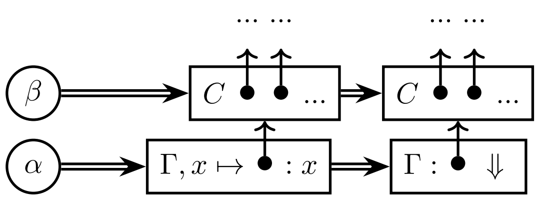


Figure 4.10: Deeply Nested Events

Note the new events corresponding to the annotated variables of the constructor application. Concerning causal connections we can easily repeat the process from above on any level to see that the "obvious" approach is, in fact, the correct one: Whenever a rule just passes through a term without making any assumptions about its contents, we are allowed to just "copy" all nested cause annotations along with it.

In order to save space in event diagrams, we will notate this as follows:

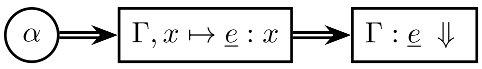

α
$\Gamma, x \mapsto \underline{e} : x$
$\Gamma : \underline{e} \Downarrow$

Figure 4.11: Deeply Nested Events Notation

Formally this means that we implicitly assume the existence of sub-events corresponding to all nested annotated terms. Furthermore, where there is a causal dependency between two events mentioning the same annotations as shown above, the two implicit event trees should be seen as connected at every node.

Finally note once more that this *only* applies where a term simply gets passed through. Once we have an event that actually depends both on the contents as well as the container this forces us to "merge" the causes. In our real-world example, actually reading a letter sent by post depends on both the letter as well as the contents at the same time. For our cost semantics, this might be matching the constructor application rule:

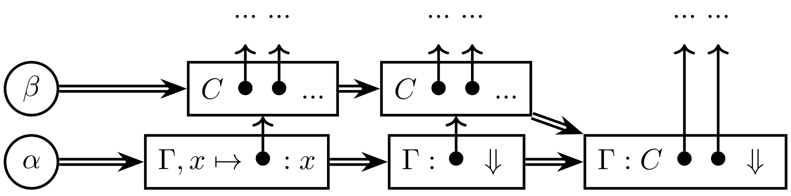


Figure 4.12: Depending on Nested Events

Note that there is a subtle difference between the events in the last two columns: While the first event represents the evaluation of an arbitrary expression, which happens to be a constructor application, the event on the right is the event of matching exactly a constructor application.

4.3.5 Variable Rule

Let us recapitulate what we learned about the variable rule up to this point. We know that analogous to Section 4.3, all events depend on matching a variable expression in the first place – if the event vanishes, so do all other events that come after it. On the other hand we know that e, θ and v get passed through no matter their value, therefore the input events that bind them cause exactly the output events that use them. This especially applies to nested events corresponding to annotated sub-terms.

Figure 4.13 shows the causal diagram with all the mentioned connections. For illustration we have included one level of nested events corresponding to the values of e and v. Note that the $\underline{e}$ expression has further nested annotations, therefore there

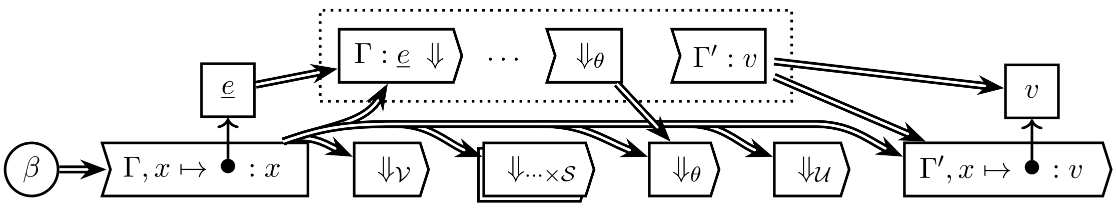

$\Gamma : \underline{e} \Downarrow$
$\cdots$
$\Downarrow_\theta$
$\Gamma' : v$
$\underline{e}$
v
β
$\Gamma, x \mapsto \bullet : x$
$\Downarrow_\mathcal{V}$
$\Downarrow_{\cdots\times\mathcal{S}}$
$\Downarrow_\theta$
$\Downarrow_\mathcal{U}$
$\Gamma', x \mapsto \bullet : v$

Figure 4.13: Causal Dependencies for a Variable Rule Match

are still more nested events here that we do not show. However, it is not hard to convince ourselves that we have all causal connections covered: All three input events are connected to exactly the appropriate output events as outlined above, with nested events following along as required.

Let us now derive annotations. Note that we have very few annotations to choose from: We have identified e, θ and v as getting passed through, which means that we do not need to reason explicitly about their annotations. Therefore the only half-way interesting cause term here is the match cause β, which needs to be annotated on all output terms. This yields us the following annotated rule:

$$\frac{\Gamma : \underline{e} \Downarrow_\theta \Gamma' : \underline{v}}{\Gamma, x \mapsto \underline{e} : {}_{\langle\beta\rangle}x \Downarrow_{\langle\beta\rangle\mathcal{V}+(3+\mathrm{lv}(e))\times_{\langle\beta\rangle}\mathcal{S}+_{\langle\beta\rangle}\mathcal{U}+_{\langle\beta\rangle}\theta} \Gamma', x \mapsto \underline{v} : {}_{\langle\beta\rangle}\underline{v}}$$

At this point it should not be too hard to see how this rule corresponds to the causal network from Figure 4.13. However there are still two notable subtleties here. Let us tackle the easy on first: We are adding an β annotation on θ as well as the return event. Strictly speaking we would expect the output events of the nested rule match to already be caused by our variable match. After all, the nested rule would have had no chance to produce an output event if we had not passed it an expression to evaluate in the first place! Would this mean by transitivity that the θ cost input event is already caused by β, making the annotation redundant?

While this is a completely valid point, this was exactly what we were talking about earlier in Section 4.2.1: For causal reasoning the judgement attempt is a precondition, therefore their cause is not reflected in annotations. We can think of this in terms of scopes: Our nested rule match will only reason causally within its own scope (shown as a dotted rectangle) and expect the outer rule application to fill in the appropriate cause for the call on the return path. This is what we are doing when we put β on the returned cost: We see terms leaving the scope, and therefore annotate it with what we see as causing execution to enter the scope in the first place. This works naturally with our causal semantics rule – we just need to put an annotation on everything we return no matter where it comes from originally.

4.4 Heap

However, there is a much more unpleasant issue with what we have *not* annotated in the rule from the last section. Have a close look at the returned heap: We have not added a β annotation on the value v when we put it back on the heap. In Figure 4.13 we also neglected to connect the nested v event on the right-hand side with the variable expression match event and therefore β. What is more, we also pass the heap Γ through without modifying any annotations. Are we sure that all this is the right thing to do?

At this point it should have become a reflex to consult a Lewis-style closest-world thought experiment. If the match did not happen, what heap would we expect to see? Comparing the answer against the original world should normally guide us to the effects we are looking for with respect to the cause β. However, this time this tried-and-true recipe leads us into big trouble. Consider that according to our rule the new heap is $\Gamma', x \mapsto v$, which means that we propagate any heap changes by the nested rule, as well as adding our own update. The easiest alternate world might now be:

$$\Gamma : \bot \Downarrow_{\bot} \{\} : \bot$$

This would have us conclude that every single surviving heap cell is direct effect of us applying the rule. This is clearly going overboard, as it is pretty clear that the "default" behaviour for a rule is to not touch the heap at all.

Therefore we might propose:

$$\Gamma : \bot \Downarrow_{\bot} \Gamma : \bot$$

Which means that we reduce the effects to all heap changes that happened during the rule match. However, this especially includes every single thunk update that happened in the meantime. If we took this viewpoint systematically, this would mean that a thunk update would receive an annotation update from every single active rule match. This is clearly not in the spirit of the operational semantics – the whole point of lazy evaluation is that the location of thunk evaluation does not matter!

4.4.1 Laziness

Remember our goal with putting annotations on terms: Our intention is to track what effects their respective values will have. Normally we have to assume that any change in a term can have arbitrarily complex consequences down the line due to transitivity. Yet for heap cells we actually happen to know that they only get updated in a very specific manner. Namely, thunks only get replaced exactly by their result value, which will be the same no matter at what point we decide to perform the evaluation.

At this point we have run into the transitivity issue introduced in Section 2.3.5 on page 18: By being too greedy about identifying our effects we end up overestimating them. We should not ask whether having $\Gamma', x \mapsto v$ constitutes a heap change, but whether the update ultimately causes a change in the program's behaviour. If our intuition is correct, this viewpoint will yield us much more sensible results.

Let us then consider a somewhat different thought experiment. Imagine that in a world W' we successfully matched the variable rule, but this time miraculously neglected to perform a heap update:

$$\frac{\Gamma_0 : e \ \Downarrow_\theta \ \Gamma_1 : v}{\Gamma_0, x \mapsto e : x \ \Downarrow_{\mathcal{V}+(3+\mathrm{lv}(e))\times\mathcal{S}+\mathcal{U}+\theta} \ \Gamma_1, x \mapsto \colorbox{lightgray}{e} : v}$$

How would things differ from the events in our original world? Where should we look for the effects? Fortunately, we can easily isolate where the first effects will appear: Of all rules from our semantics in Figure 3.4 on page 55, the (Var) rules are the only ones to ever read a heap cell. Furthermore, the (Let) rule is careful to never overwrite existing heap cells by using fresh names. We can conclude that the evaluation in W' will remain virtually the same until exactly the point where the expression x gets evaluated the next time. Unfortunately, this re-evaluation might then have further side-effects on program state[4]. After all, repeated evaluation might spawn new thunks, which might in turn cause even more repeated evaluations!

4.4.2 Set-Up

Let us prove that if we miraculously forget heap updates like this, it will have no "material" effect on program execution. This property would absolve us from putting an annotation on e in the above example, as well as any other updated heap cell on Γ_1. So what would such an effect look like? As per Lewis, we are looking for changes in the alternate world W'. More specifically, we want events that happen in W but do *not* happen in our alternate world W'. It is important to note that do not care about additional events in W', as for performance analysis we are only interested in events that happen in W. In the nomenclature of Taylor [1993], we only track positive causation. It is a good idea to keep this in mind, as duplicated thunk evaluations are going to generate quite a few extra events in W'.

Furthermore, even events that exist in both W and W' will not look exactly the same: After all, we might evaluate more (Let) expressions in W', which means that we will also see more names freshly generated. And while we expect these names to refer to heap cells that have an equivalent in W, this still forces us to start reasoning in terms of equivalence. Let us assume that we have a partial function f mapping variable

[4]As Marlow and Peyton Jones [2011] remark, laziness ironically causes plentiful side-effects.

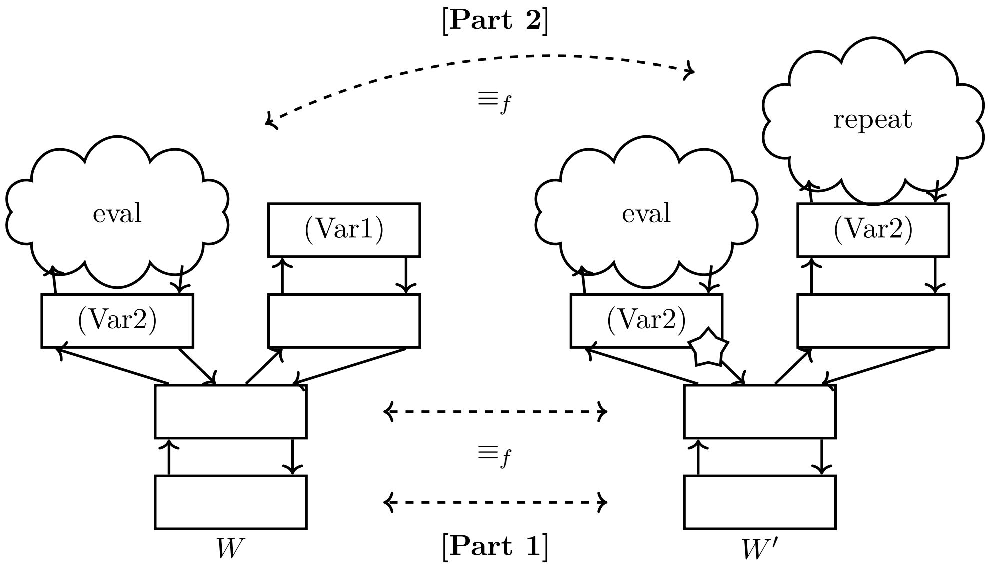


Figure 4.14: Laziness Transparency Proof Overview

names from W to either the name of their clone in W' or $\bot$ otherwise. Then we define equivalence on expressions as follows:

$$\begin{array}{rcll}
x & \equiv_f & x' & \text{where } x = x' \text{ or } f(x) = x' \\
C\ x_1 ... & \equiv_f & C\ x'_1 ... & \text{where } x_1 \equiv_f x'_1, ... \\
\lambda y.e & \equiv_f & \lambda y.e' & \text{where } e \equiv_f e' \\
e\ x & \equiv_f & e'\ x' & \text{where } e \equiv_f e' \text{ and } x \equiv_f x' \\
\texttt{let}\ \{x_i = e_i, \ldots\}\ \texttt{in}\ e & \equiv_f & \texttt{let}\ \{x_i = e'_i, \ldots\}\ \texttt{in}\ e' & \\
 & & & \text{where } e_i \equiv_f e'_i, e \equiv_f e', ... \\
\texttt{case}\ e\ \texttt{of}\ \{C\ x_{ij} ... \to e_i; \ldots\} & \equiv_f & \texttt{case}\ e'\ \texttt{of}\ \{C\ x_{ij} ... \to e'_i; \ldots\} & \\
 & & & \text{where } e \equiv_f e', e_i \equiv_f e'_i, ...
\end{array}$$

and equivalence on heaps along the same lines:

$$\begin{array}{rcll}
\Gamma, x \mapsto e & \equiv_f & \Gamma', x \mapsto e' & \text{where } \Gamma \equiv_f \Gamma', e \equiv_f e' \text{ and } f(x) \notin \Gamma \\
\Gamma, x \mapsto e & \equiv_f & \Gamma', x \mapsto e', f(x) \mapsto e'' & \text{where } \Gamma \equiv_f \Gamma' \\
 & & & \text{and } \begin{cases} e \equiv_f e' \text{ or } e \equiv_f e'' & \text{if } e \text{ whnf} \\ e \equiv_f e' \text{ and } e \equiv_f e'' & \text{otherwise} \end{cases} \\
\{\} & \equiv_f & \{\} &
\end{array}$$

Take note how we treat duplicated heap cells for the W' heap: Neither copy is allowed to be further evaluated than the original, and if the original world sees the expression e in weak normal form, at least one of the copies must be evaluated as well.

As shown in Figure 4.14, our proof will consist of two parts: First we need to show that until we evaluate one of the "duplicated" variables from f, evaluation in W and W' will proceed in lock-step: All events that happened in W should find their equivalent in W'. However once we hit one of the duplicated variables, our task changes: At this point the original world W will simply return the result, whereas W' repeats the evaluation. As there are no equivalent events in W we do not care about how duplicated evaluation in W' looks like, but we need to prove that we arrive at an equivalent return value and heap in order to resume synchronised evaluation.

4.4.3 Proof Part 1

Due to the nature of reasoning about evaluation using nested rule matches, both parts of the proof will take the form of an induction over the rule match tree. Assume that we have a mapping f and two rule matches in W and W' respectively:

$$\Gamma_a : e \Downarrow_\theta \Gamma_b : v \quad \text{and} \quad \Gamma'_a : e' \Downarrow_{\theta'} \Gamma'_b : v'$$

with $\Gamma_a \equiv_f \Gamma'_a$ and $e \equiv_f e'$. Our induction assumption is that we will be able to build a new refined mapping g with $g(x) = f(x)$ where $f(x) \neq \bot$ such that $\Gamma_b \equiv_g \Gamma'_b$ and $v \equiv_g v'$. Note that this implies $\Gamma_a \equiv_g \Gamma'_a$ and $e \equiv_g e'$. We can prove this trivially if e happens to be a constructor or lambda expression: We know due to $e \equiv_f e'$ that we must find an equivalent expression in W', and the structure of the rules allows us to simply set the new mapping $g = f$ to obtain $\Gamma_b \equiv_g \Gamma'_b$ and $v \equiv_g v'$.

However things gets slightly more complicated for other rules. Suppose we find an application expression in W:

$$\frac{\Gamma_a : e \Downarrow_{\theta_f} \Gamma_b : \lambda y_1.\lambda y_2....e_b \qquad \Gamma_b : e_b[x_i/y_i \cdots] \Downarrow_{\theta_b} \Gamma_c : v}{\Gamma_a : e\ x_1\ x_2 \cdots \Downarrow_{...+\theta_f+\theta_b} \Gamma_c : v}$$

Again we know that in W' we must also have encountered an equivalent application expression, and that $\Gamma_a \equiv_f \Gamma'_a$ and especially $e \equiv_f e'$. Therefore it follows by induction that we can obtain a new mapping g from the nested rule match, with $\Gamma_b \equiv_g \Gamma'_b$ and $\lambda y_1.\lambda y_2....e_b \equiv_g \lambda y_1.\lambda y_2....e'_b$. As this implies $e_b \equiv_g e'_b$ and we know that g is constructed such that $x_i \equiv_g x'_i$ it follows that $e_b[x_i/y_i] \equiv_g e'_b[x'_i/y_i]$ [5]. Using our induction assumption on the second nested rule match now yields us a final mapping h, which must satisfy $\Gamma_c \equiv_h \Gamma'_c$ and $v \equiv_h v'$, which is exactly what we need to be finished with this case.

[5]Note that this is only true if the substitution never changes a name bound by a lambda, `let` expression or `case` expression. This has been proven by Sestoft [1997, section 2.5].

For `let` expressions we would have the following W judgement:

$$\frac{\Gamma_a, y_i \mapsto e_i[y_i/x_i, ...], ... : e[y_i/x_i, ...] \ \Downarrow_{\theta_l} \ \Gamma_b : v}{\Gamma_a : \texttt{let}\ \{x_i = e_i, ...\}\ \texttt{in}\ e \ \Downarrow_{...+\theta_l} \ \Gamma_b : v}$$

As before we will have an equivalent W' expression of the form $\texttt{let}\,\{x_i = e'_i, ...\}\,\texttt{in}\ e'$. With no loss of generality we assume that in both W and W' we choose the same fresh variables y_i to refer to the allocated heap cells. As we also know that the x_i will be the same in W and W' it follows trivially that $e[y_i/x_i] \equiv_f e'[y_i/x_i]$. As this extends to the heap cells we also have $\Gamma_a, y_i \mapsto e_i[y_i/x_i, ...], ... \equiv_f \Gamma'_a, y_i \mapsto e'_i[y_i/x_i, ...], ...$, at which point our hypothesis follows from the induction assumption.

Up to this point tackling rules has been pretty mechanical, as we have of course deliberately defined equivalence so terms can only differ in variable names. The most interesting part of this proof is actually, again, the variable evaluation rule. Let us suppose we find the variable in question unevaluated in W. This would prompt us to match the (Var2) rule:

$$\frac{\Gamma_a : e \ \Downarrow_\theta \ \Gamma_b : v}{\Gamma_a, x \mapsto e : x \ \Downarrow_{...+\theta} \ \Gamma_b, x \mapsto v : v}$$

As before we must also encounter a variable expression in the alternate world. However, due to duplication we can find two distinct concrete variables x' in W': The choice is between $x' = x$ and $x' = f(x)$. However, it follows directly from $\Gamma_a \equiv_f \Gamma'_a$ that neither can be evaluated on Γ'_a. To be specific, we know that we will find an unevaluated thunk e' with $e \equiv_f e'$. Therefore we can again conclude by induction that the nested rule matches will yield us a new mapping g complete with new heaps $\Gamma_b \equiv_g \Gamma'_b$ and result $v \equiv_g v'$. If we recall our heap equivalence definition, it is also not hard to convince ourselves that both $\Gamma_b, x \mapsto v \equiv_g \Gamma'_b, x \mapsto v'$ as well as $\Gamma_b, x \mapsto v \equiv_g \Gamma'_b, f(x) \mapsto v'$ holds, therefore we have heap equivalence no matter whether we have $x' = x$ or $x' = f(x)$.

Things start to diverge a bit more if the heap cell happens to be evaluated in W. This would prompt us to match the (Var1) rule:

$$\Gamma, x \mapsto v : x \ \Downarrow_{\mathcal{V}+\mathcal{S}} \ \Gamma, x \mapsto v : v$$

It is easy to prove that the desired properties hold if we assume that the variable in question is also evaluated in W'. However this is not guaranteed, as the prior (Var2) rule match only ever updates one of x or $f(x)$ in W'. This means that the x' heap cell might turn out to not be evaluated, which matches (Var2) in W':

$$\frac{\Gamma'_a : e' \ \Downarrow_{\theta'} \ \Gamma'_b : v'}{\Gamma'_a, x' \mapsto e' : x' \ \Downarrow_{...+\theta'} \ \Gamma'_b, x' \mapsto v' : v'}$$

4.4.4 Proof Part 2

This is the interesting spot: Matching a different rule means that the histories of W and W' are going to differ at this point. Our goal is to show that we can restore lock-step evaluation between the two worlds in a way that satisfies our notion of equivalence. First note that we know at this point that world W must have seen a (Var2) rule match on x in the past. This is because we know that only `let` expressions add new bindings to heap, so even if it added a binding in normal form to the heap, it would be in normal form in both W and W', making it impossible for the (Var2) to ever match it. Hence the heap cell must have seen an update at some point W^* in the past:

$$\frac{\Gamma_a^* : e^* \ \Downarrow_{\theta^*} \ \Gamma_b^* : v}{\Gamma_a^*, x \mapsto e : x \ \Downarrow_{\ldots+\theta^*} \ \Gamma_b^*, x \mapsto v : v}$$

Even though this event might have happened at an arbitrary point in the past, there are a few things we know: The variable matched must also have been x, and the evaluation result must have been v, as no heap cell ever gets updated twice. A little less obvious is that we also have $e^* \equiv_f e'$. This is due to fact that the only way an unevaluated e^* expression could exist on the heap of W is due to a `let` expression binding it. At that point we know that W' should bind x as well, so equivalence is guaranteed for $x' = x$. And as we will see, we will later guarantee equivalence for when we bind $f(x)$ as well.

Let us then get back to the duplicated thunk evaluation of e'. As mentioned earlier, this evaluation might spawn additional thunks, which we want to associate with the "original" bindings in W using our mapping f. We do this by – again – comparing the rule match tree of W' against what happened in W. Only this time we actually compare against a different part of W: The point of its history where it originally evaluated x. We will call this part W^* from now on to contrast it from the "present" events in W. This means that we see the W heap Γ as fixed for now and assume two parallel rule matches in W^* and W':

$$\Gamma_a^* : e^* \ \Downarrow_{\theta^*} \ \Gamma_b^* : v^* \quad \text{and} \quad \Gamma_a' : e' \ \Downarrow_{\theta'} \ \Gamma_b' : v'$$

Again we know that we should have $e^* \equiv_f e'$, and want to derive a new mapping g satisfying at least $v^* \equiv_g v$. This time around we are treating the heap differently though: As it does not change in the original evaluation in W, we need to produce a mapping g such that $\Gamma \equiv_g \Gamma_a'$ as well as $\Gamma \equiv_g \Gamma_b'$. Or put in other words: Every new binding we add in W' during this evaluation *must* be a duplicate of an existing binding in W^*. Note that we do not have any proof obligations regarding Γ_a^* or Γ_b^*.

We can prove this using a familiar strategy: Using induction we walk the rule match trees of W^* and W' in parallel, upholding our new induction assumption at every step.

This allows us to easily knock down variables, lambda as well as application expressions. Let us then assume that we encounter a `let` expression:

$$\frac{\Gamma_a^*, y_i \mapsto e_i^*[y_i/x_i, ...], ... : e^*[y_i/x_i, ...] \Downarrow_{\theta_l^*} \Gamma_b^* : v^*}{\Gamma_a^* : \texttt{let}\ \{x_i = e_i^*, ...\}\ \texttt{in}\ e \Downarrow_{...+\theta_l^*} \Gamma_b^* : v^*}$$

Note that the induction proof in the first part of the proof covered the entire rule match tree of W, and therefore especially the part that we now refer to as W^*. This means that back then we must also have added the y_i bindings to the past heap in W'. Consequently y_i are no longer fresh in W', so we are forced to choose new names y_i'. In order to track this duplication, we now need to perform our first update to the map, setting $g(y_i) = y_i'$ and $g(y) = f(y)$ otherwise. We know that $f(y_i) = \bot$ because `let` expressions will remain the only instance where we update the map, and we will only ever duplicate the evaluation of a `let` expression once. After all, an expression only can get evaluated twice if it a thunk in W, and heap equivalence guarantees that for every thunk in W there can only be either one or two thunks in W'.

The updated map establishes $y_i \equiv_g g(y_i)$, so once again we have $e^*[y_i/x_i] \equiv_g e'[g(y_i)/x_i]$ as well as $e_i^*[y_i/x_i] \equiv_g e_i'[g(y_i)/x_i]$. Furthermore, we know that updating our heap does not violate equivalence, so we still have $\Gamma \equiv_g \Gamma_a', g(y_i) \mapsto e_i'[g(y_i)/x_i, ...]$: In case Γ still maps y_i to an unevaluated expression we know it to be $e_i^*[y_i/x_i]$, which must be equivalent to both the existing y_i mapping in Γ_a' as well as the new $g(y_i)$ mapping as established. On the other hand if y_i is evaluated on Γ we have no further demands to satisfy. Therefore the heap passed to the nested judgement satisfies our induction hypothesis, so we can obtain another mapping h for the nested rule match satisfying $\Gamma \equiv_h \Gamma_b'$ and $v^* \equiv_h v'$, which is what we wanted to show.

However, we still have to take care of the variable rule. After all, our current repeated evaluation might spawn *further* repeated evaluations, which is why we are putting this much effort into this to begin with. We know due to expression equivalence that where W^* has a variable term x^* we will get a variable x' in W', where either $x' = x^*$ or $x' = g(x^*)$. Let us first consider the case that we find x' to be already evaluated on Γ_a':

$$\Gamma_a', x' \mapsto v' : x' \Downarrow_{\mathcal{V}+\mathcal{S}} \Gamma_a', x' \mapsto v' : v'$$

Note that we are now splitting up the cases by what we find in W', as that is more convenient in this instance. As we require that $\Gamma \equiv_f \Gamma_a'$ this means that x^* must also be evaluated on Γ. Furthermore the value must be v^*, as both (Var1) and (Var2) return the value that was pushed on the heap Γ_b^*, and as Γ was derived from it we know that the same value must still be present. Finally we know that $v^* \equiv_f v'$ and trivially $\Gamma_b^* \equiv \Gamma_b'$, which is our final proof obligation for this case.

This leaves only the case that the variable x' happens to refer to a thunk in W', which would mean that we match the (Var2) rule instead:

$$\frac{\Gamma'_a : e' \;\Downarrow_{\theta'}\; \Gamma'_b : v'}{\Gamma'_a, x' \mapsto e' : x' \;\Downarrow_{\ldots+\theta'}\; \Gamma'_b, x' \mapsto v' : v'}$$

Now we again have two cases: If the original-world variable x^* also had to be evaluated in W^*, we can easily use the induction hypothesis to show that we can obtain a new mapping g with $v^* \equiv_g v'$ and $\Gamma'_b \equiv_g \Gamma$, from which we can easily follow through. On the other hand if x^* turns out not to be evaluated, we use the same argument as we did in part one: Clearly this is another duplicated evaluation, therefore we simply need to compare our current W' with another past evaluation W^{**}. We would still reason about the same fixed W heap Γ, so our induction hypothesis would still apply even though we have switched out the concrete rule application tree.

It is clear that we can do this recursively: By always jumping to the appropriate past evaluation point in W, we will always find a way to continue the induction. Especially note that we are always going backwards in the (finite) rule application history – so we are not attempting to inductively reason about an infinite tree here.

4.4.5 Wrapping Up

At this point we have set up all the most important ingredients to our proof. We just need to plug a few holes to put everything together. First note that we have ignored `case` expressions so far, which are however straight-forward to add. More interestingly, we have left no place where we could introduce our "miracle" into the proof. In fact due to the lock-step evaluation between W and W', it might seem like we could not introduce any change to W' at any point. This was deliberate, as this means that we can now introduce it at an arbitrary point.

Let us assume that at some point in the first part of the proof, but have not yet encountered a duplication. This means that $f(y) = \bot$ for all y, from which by induction hypothesis follows that $\Gamma_a = \Gamma'_a$ and $e = e'$. Suppose the judgement will look like follows:

$$\Gamma_a : e \;\Downarrow_\theta\; \Gamma_b : v$$

Normally we would expect that we would derive the same heap $\Gamma_b = \Gamma'_b$ in both worlds as well. But now suppose that we choose an arbitrary binding x from the heap Γ_a before evaluation and "forget" its update. If put as a rule, it would look like follows:

$$\frac{\Gamma_a; x \mapsto e_0 : e \;\Downarrow_\theta\; \Gamma_b, x \mapsto e_1 : v}{\Gamma_a; x \mapsto e_0 : e \;\Downarrow_\theta\; \Gamma_b, x \mapsto e_0, g(x) \mapsto e_1 : v}$$

With $g(x)$ fresh and $g(y) = \bot$ otherwise. Note that additionally to forgetting about x, the $g(x)$ binding now refers to an (inaccessible) copy of e_1. This might seem like an odd choice, but this trick means that for $\Gamma'_b = \Gamma_b, x \mapsto e_0, g(x) \mapsto e_1$ we now have $\Gamma_b \equiv_g \Gamma'_b$. As furthermore $v \equiv_g v'$ this means that we have not violated the induction hypothesis, which means that the rest of the proof remains valid. Therefore – as long as we start in the same state and introduce one miracle at maximum, we know that we can retain synchronisation between the two worlds.

We demonstrate that W and W' proceed synchronously so we can argue about the underlying events, which allows us to identify effects – or preferably the lack thereof. The idea is that in part one we mapped every non-variable rule application in W to exactly the same rule matching in W' *and* established that there is an equivalence relation between the used terms. As we explained in Section 4.2 on page 59 and Section 4.3.4 on page 68, every rule match translates systematically to a set of events, therefore we know that the arising events must be equivalent as well. The only exception is when we match $(Var1)$ in W, but $(Var2)$ in W', but even then we can easily see that $(Var1)$ produces strictly less events than $(Var2)$. Note that this is especially true for cost events: Contrasting the cost terms emitted by the two variable evaluation rules we see that indeed $\mathcal{V}+\mathcal{S} < \mathcal{V}+(3+\mathrm{lv}(e))\times\mathcal{S}+\mathcal{U}+\theta$. Note that this property is actually vital for this argument to work – if looking up heap values could possibly be more costly than evaluation we would have positive causation of cost with the heap update!

The way this proof has worked out, we see that replacing a thunk by its normal-form evaluation result will never produce less costs[6]. Therefore there are no effects left to track, and we can safely omit annotations on updates such as the one in Section 4.3.5. Note that this is especially true if our rule match "inherits" the thunk update in question from a nested rule match. Together with the annotation encapsulation property that we are going to show later in Section 4.7.1, this allows us to pass through updated heaps from nested rule matches without the need for annotation updates.

4.5 Interrupted Rules

At this point we know how to annotate basic nested rules: Annotations are required on all results caused by the rule match, excluding heap updates. However, what happens if we have more than one nested rule match, such as with function applications? Recall from Figure 3.4 that we gave the following semantics:

$$\frac{\Gamma : e \Downarrow_{\theta_f} \boxed{\Gamma' : \lambda y_1.\lambda y_2.\ldots.e_b} \qquad \Gamma' : e_b[x_i/y_i\cdots] \Downarrow_{\theta_b} \Gamma'' : v}{\boxed{\Gamma : e\ x_1\ x_2\cdots} \Downarrow_{\mathcal{A}+(1+n+\mathrm{lv}(e))\times\mathcal{S}+\theta_f+\theta_b} \Gamma'' : v}$$

[6] Note that if we tracked residency costs as mentioned in Section 5.2.5 on page 120, there could be scenarios where keeping the normal-form result around is more expensive than reverting to the thunk. However, we regard these cases as too obscure to be relevant for profiling.

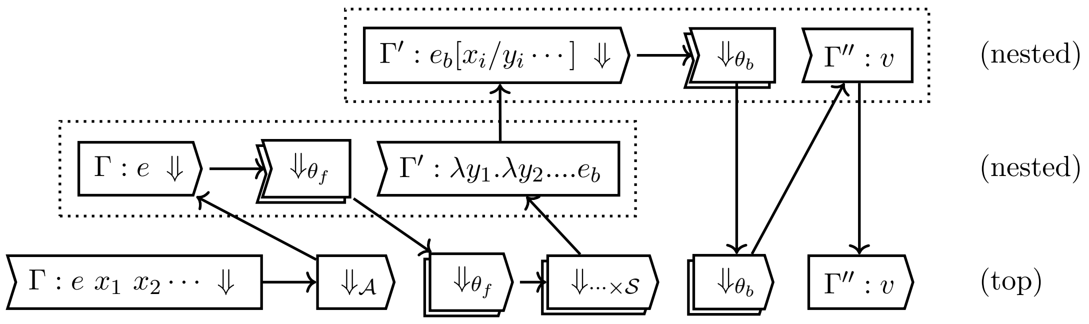

$\Gamma' : e_b[x_i/y_i \cdots] \Downarrow$
$\Downarrow_{\theta_b}$
$\Gamma'' : v$
(nested)
$\Gamma : e \Downarrow$
$\Downarrow_{\theta_f}$
$\Gamma' : \lambda y_1.\lambda y_2....e_b$
(nested)
$\Gamma : e\ x_1\ x_2 \cdots \Downarrow$
$\Downarrow_{\mathcal{A}}$
$\Downarrow_{\theta_f}$
$\Downarrow_{\cdots \times \mathcal{S}}$
$\Downarrow_{\theta_b}$
$\Gamma'' : v$
(top)

Figure 4.15: Ordered Event Decomposition of an Application Rule Match

The high-level view is that after matching the application $e\ x_1\ x_2 \cdots$ we first evaluate the function expression e, expecting a function body $\lambda y_1.\lambda y_2....e_b$, nested inside an appropriate number of lambda terms. We then proceed to bind parameters by substituting the variables in e_b and evaluate the result to attain the result value, which we pass through. We can decompose this process into events as shown in Figure 4.15. Note that we want to be specific about the order in which costs appear: We assume that constant costs happen first, and that we push the parameters on the stack before jumping to the closure code. This will become important in a moment.

For causality analysis we now again ask how the events of the rule match could be disrupted within the constraints of our language and judgement form. As highlighted in the above rule, our rule makes two assumptions about the input events that could turn out to be false. First we assume again that the expression has the form we expect – a function application. More interestingly, this time we also assume that evaluation of the function expression yields us a suitable lambda expression. All remaining input events will not cause the rule to mismatch, and therefore get passed through as explained in Section 4.3.2.

Let us consider the expression under evaluation first, as it comes first according to our temporal order. Once more we apply our basic recipe of substituting $\perp$ for forcing an expression match failure in the alternate world W':

$$\Gamma : \perp \Downarrow_{\perp} \Gamma : \perp$$

It might seem redundant to re-state this, but we need to remember that we always select this alternate world by choice. We will see in Section 4.6 on page 85 that for some expression types it makes sense to choose differently. However, this time we obtain the usual result: all events vanish. We are left with a frozen program and the finding that all later rule match events causally depend on the initial match happening.

4.5.1 Miraculous Interruption

However, let us turn our attention to the point of interest for this rule: The effects of a mismatch on the lambda value that supposedly results from evaluation. It might seem like the type-checker should assure this property – at least ignoring termination – but as explained in Section 4.2, in alternate worlds we are working with miracle events that are allowed to momentarily ignore such consistency requirements. In our case, what we want is another fresh world W'' where we did find a function application, yet the returned expression magically turned out to *not* have the form $\lambda y_1.\lambda y_2....e_b$. Again there is nothing that would strongly suggest otherwise, so let our "miracle" be that the return value for some reason turns out to be "$\perp$", causing the match to fail.

This leads to a small problem: When consulting Figure 3.4 we see that no rule would match a result of $\perp$. This is not surprising, as provoking a mismatch was the whole point of this exercise. Accordingly, we would be forced to judge:

$$\Gamma : \perp \ \Downarrow_{\perp} \ \Gamma : \perp$$

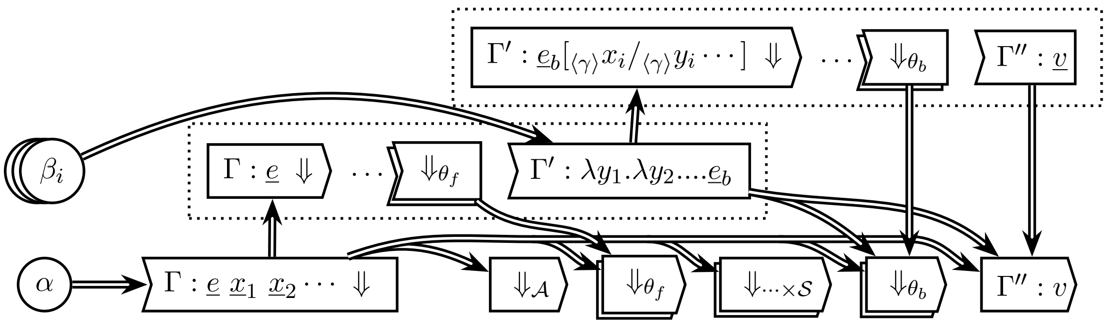
Yet there are strong reasons why we cannot accept this. Take note that this alternate world judgement would eradicate *all* events shown in Figure 4.15, and therefore especially events that *precede* the miracle result event in the temporal order. Therefore the miracle event has actual effects that lie in its own past, which we outlawed back in Section 4.2 on page 59.

So what can we do? We have to find some meaningful way to continue writing the story of our alternate world W''. Let us think back a bit about how a real Haskell program might run through the motions in our alternate world W'': It would actually evaluate the body, triggering all associated events, proceed with pushing all required parameters, then attempt to jump to the sneakily placed non-closure and – presumably – crash. Or put in other words: All events prior to the failed match would happen, but none of the events after that point. Not coincidentally, this also happens to be the minimal solution satisfying our requirements for causality reasoning. We therefore propose the following additional rule for W'' as the closest-possible description of the last breaths of our crashing program[7]:

$$\frac{\Gamma : e \ \Downarrow_{\theta_f} \ \Gamma' : \perp}{\Gamma : e \ x_1 \ x_2 \cdots \ \Downarrow_{\mathcal{A}+(1+n+\mathrm{lv}(e))\times\mathcal{S}+\theta_f+\perp} \ \Gamma' : \perp}$$

Consequently, we conclude that the successful match in the original world W causes exactly all rule events to happen that come *after* it in the temporal order.

[7]The argument would be cleaner using small-step semantics, as they correspond more closely with the sketched program behaviour. However, this would make working with the semantics harder overall, so we choose this path instead.



$\Gamma' : \underline{e}_b[{}_{\langle\gamma\rangle}x_i/{}_{\langle\gamma\rangle}y_i \cdots] \Downarrow$
$\cdots$
$\Downarrow_{\theta_b}$
$\Gamma'' : \underline{v}$
β_i
$\Gamma : \underline{e} \Downarrow$
$\cdots$
$\Downarrow_{\theta_f}$
$\Gamma' : \lambda y_1.\lambda y_2 \ldots . \underline{e}_b$
α
$\Gamma : \underline{e}\ \underline{x}_1\ \underline{x}_2 \cdots \Downarrow$
$\Downarrow_{\mathcal{A}}$
$\Downarrow_{\theta_f}$
$\Downarrow_{\cdots \times \mathcal{S}}$
$\Downarrow_{\theta_b}$
$\Gamma'' : v$

Figure 4.16: Causal Relations for an Application Rule Match

4.5.2 Application Rule

At this point we have collected enough information to complete our picture of the causal relationships within the application rule. The result is shown in Figure 4.16: We have five "input" event types that can fail due to outside influences and variously cause five "output" events that are visible to the rule caller. We can summarise as follows:

Cause	Input Event	Causal Relationship				
α	$\Gamma : \underline{e}\ \underline{x}_1\ \underline{x}_2 \cdots \Downarrow$	✓	✓	✓	✓	✓
	$\Downarrow_{\theta_f}$		✓			
β_i	$\Gamma' : \lambda y_1.\lambda y_2 \ldots . \underline{e}_b$			✓	✓	✓
	$\Downarrow_{\theta_b}$				✓	
	$\Gamma'' : \underline{v}$					✓
	Output Events:	$\Downarrow_{\mathcal{A}}$	$\Downarrow_{\theta_f}$	$\Downarrow_{\ldots \times \mathcal{S}}$	$\Downarrow_{\theta_b}$	$\Gamma'' : \underline{v}$

Figure 4.17: Summary of Causality Relations

We observe that in two instances input events are required for the rule to continue at all, while the remaining causes just represent terms getting passed through according to Section 4.3.2 on page 66. We see the profile events as representing all cost events emitted by the first and second nested rule match respectively.

Let us get back to the original question: Exactly what annotations should we put on the terms represented by the "output" events? We know the causes that were annotated on the input terms, so we apply the transitive property: If what caused the input event

"$\Gamma : \underline{e}\ \underline{x}_1\ \underline{x}_2 \cdots\ \Downarrow$ " is described by the cause term α, then according to our causality diagram this is exactly the cause term we should, say, annotate on the $\mathcal{A}$ unit of cost emitted, yielding ${}_{\langle\alpha\rangle}\mathcal{A}$. Along the same lines we need to annotate the cause on all cost getting passed through in θ_f, which we would notate as ${}_{\langle\alpha\rangle}\theta_f$.

The rest of the output events have a causal dependency with the lambda result coming from the evaluation of the function expression. Especially note that the pattern "$\Gamma' : \lambda y_1.\lambda y_2....\underline{e}_b$" matches the expression deeply, as we need to substitute *all* bound variables in order to proceed with evaluation. As explained in Section 4.3.4 on page 68 this means that we are actually looking through a number of annotations, which means that we are actually reasoning about a number of causes at the same time:

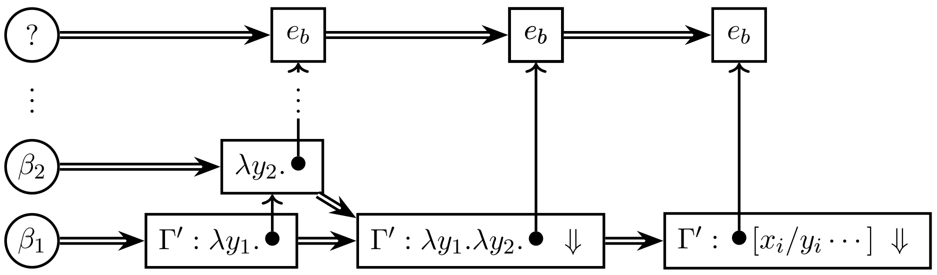

?
β_2
β_1
e_b
e_b
e_b
$\lambda y_2.$
$\Gamma' : \lambda y_1.$
$\Gamma' : \lambda y_1.\lambda y_2.\ \Downarrow$
$\Gamma' : [x_i/y_i \cdots]\ \Downarrow$

Figure 4.18: Depending on Deeply Nested Annotations

Only once we have looked through enough lambda terms we reach an the term $\underline{e}_b$ which our rule does not have a causal dependency on anymore. In order to annotate the composite cause in a compact way, we will be making use of the following notation:

$$\beta_1 \wedge \beta_2 \wedge ... = \bigwedge_i \beta_i =: \overline{\beta_i}$$

Furthermore note that for the variable substitution we used the $\underline{e}_b[{}_{\langle\gamma\rangle}x_i/{}_{\langle\gamma\rangle}y_i \cdots]$ syntax in Figure 4.16. This simply means that instead of copying the entire tree of sub-events as explained in Section 4.3.4 on page 68, we perform the given substitution on the way. As we retain annotations, events are still in a causal relationship even after variable substitution.

All this allows us to finish up the annotated application rule:

$$\frac{\Gamma : \underline{e}\ \Downarrow_{\theta_f}\ \Gamma' : {}_{\langle\beta_1\rangle}\lambda y_1.{}_{\langle\beta_2\rangle}\lambda y_2....\underline{e}_b \qquad \Gamma' : \underline{e}_b[{}_{\langle\gamma\rangle}x_i/{}_{\langle\gamma\rangle}y_i \cdots]\ \Downarrow_{\theta_b}\ \Gamma'' : \underline{v}}{\Gamma : {}_{\langle\alpha\rangle}(\underline{e}\ \underline{x}_1\ \underline{x}_2 \cdots)\ \Downarrow_{{}_{\langle\alpha\rangle}\mathcal{A}+(1+n+\mathrm{lv}(e))\times{}_{\langle\alpha\rangle}\mathcal{S}+{}_{\langle\alpha\rangle}\theta_f+{}_{\langle\alpha\overline{\beta_i}\rangle}\theta_b}\ \Gamma'' : {}_{\langle\alpha\overline{\beta_i}\rangle}\underline{v}}$$

Just like when we settled on the annotated variable rule in Section 4.3.5 on page 70, most annotations again get passed around implicitly. The only annotations that we need to mention are the ones that correspond to actual mismatch events, which in our case are α and $\overline{\beta_i}$ respectively.

4.6 Closest World Choice

The last sections should have demonstrated that reasoning about causality is often about choosing the right viewpoint amongst several possible options. Our goal is always to find a way to see things that allows the sharpest diagnosis, meaning that we name as few effects as possible that could just be chalked up as happening independent of their supposed cause. In this section we will approach another instance where it pays to take a fresh look at how we derive causality. Specifically, we will be looking for a good way to annotate the `let` rule:

$$\frac{\Gamma, y_i \mapsto e_i[y_i/x_i, ...], ... : e[y_i/x_i, ...] \;\Downarrow_\theta\; \Gamma' : v}{\Gamma : \texttt{let}\ \{x_i = e_i, ...\}\ \texttt{in}\ e \;\Downarrow_{\sum(\mathcal{S}+\mathcal{O}(x_i=e_i))+\theta}\; \Gamma' : v}$$

On the surface, nothing here is too surprising: We are passing a number of terms to a nested rule match. As before, we would normally look for effects at the point where the rule generates its output events, comparing it against a world where we got a $\bot$ term in place of the `let` expression, yielding the familiar “crash” judgement in W':

$$\Gamma : \bot \;\Downarrow_\bot\; \Gamma : \bot$$

Following this line of reasoning, we would again put annotations on every single cost as well as the result value. But is that the right thing to do?

Note that all rules up to this point had heavy influences on the control flow of the program. The `let` expression on the other hand is primarily concerned with updating the heap, and barely touches the expression e, value v and cost θ as they pass through. After all, the primary objective of this expression is updating the heap, and at what point we do this does not actually matter too much in the grand scheme of things. In fact, optimising compilers often exploit this property by allowing `let` bindings to “float” freely within the optimised expressions [Peyton Jones et al., 1996]. We saw an example for this back in Section 3.4.2.

So could we find a way to have the program degrade more “gracefully” in the presence of our miracle? Would it, say, make sense to just “float” the `let` binding into nothingness and evaluate only its body in its stead? This would make our miraculous rule match “$\Gamma : e \;\Downarrow$ ”, ideally yielding a new W'' judgement of the form:

$$\Gamma : e \;\Downarrow_\theta\; \Gamma' : v$$

This judgement means that we attempt to match the same rule(s) in W' that the nested judgement in the (Let) rule originally did in W. The only difference is that our alternate world lacks all the bindings introduced by the `let` expression. Obviously this might cause the program to crash at a point further down the line, as leaving free variables un-renamed deliberately breaks the consistency of the semantics given by Sestoft [1997]. Yet the crash is not guaranteed – the new world W' might see the program continue normal operation for some time longer, and possibly even terminate. Therefore our new definition of W' is actually *closer* to the original world W than an immediate program crash, and therefore preferable according to Lewis' causality model.

4.6.1 Floating Effects

Let us be a bit more formal about this. We will have W as the original world and W' as our first alternate world where we simply have the program crash immediately when attempting the (Let) rule match in question. On the other hand, W'' will be our new alternate world where we simply evaluate the body instead of the full `let` expression. Our claim is that W'' has as least as many events in common with W as W', and can therefore plausibly be described as "closer". Note that this is actually not a particularly hard requirement to fulfil: The rule mismatch in W' will cause parent rule matches to fail, which as we already noted back in Section 4.3 on page 64 pretty much means that no further events are happening. From that point of view, the miracle event is the last actual event happening in W'. It follows that if W'' can possibly reproduce *any* W event past the miracle event, we have already shown the above-mentioned property.

Having said that, we are actually very interested in exactly how far we can make W and W'' match up: After all, this determines the effects and therefore the annotations. To do this we first need to recognise that due to skipping the renaming step W'' will have different variable names compared to W. So again we have to employ a notion of event equivalence. Fortunately we can simply use the same equivalence operator $\equiv_f$ we used back in Section 4.4.2, except this time we set the mapping function upfront as exactly $f(y_i) = x_i$. We also define heap equivalence as follows:

$$\begin{aligned}
\Gamma, y \mapsto e \equiv_f \Gamma', y \mapsto e' &\quad \text{where } \Gamma \equiv_f \Gamma', e \equiv_f e' \\
\Gamma, y \mapsto e \equiv_f \Gamma' &\quad \text{where } f(y) \neq \bot \\
\{\} \equiv_f \{\} &
\end{aligned}$$

Capturing that we expect the y bindings to be gone in W'.

Let us then do another induction over the rule match trees of W and W'' in parallel. Our current judgements will be $\Gamma_a : e \Downarrow_\theta \Gamma_b : v$ in W and $\Gamma''_a : e'' \Downarrow_{\theta''} \Gamma''_b : v''$ in W'', with the assumptions being $\Gamma_a \equiv_f \Gamma''_a$ and $e \equiv_f e''$. What we want to prove is that where a rule matches we get a new heap $\Gamma_b \equiv_f \Gamma''_b$ and value $v \equiv_f v''$. Note that we do not *require* the rule match to succeed in W'', as any possibly successful rule match would already mean more events happening than in W'.

We can easily see this for either a constructor or a lambda expressions in W, for example: Due to equivalence we encounter the same type of expression in W'', and the equivalence of input terms directly proves that all result terms must be equivalent as well. The remaining rules from Figure 3.4 need a bit more consideration. Let us run through them once more, starting with the application rule:

$$\frac{\Gamma : e \Downarrow_{\theta_f} \Gamma' : \lambda y_1.\lambda y_2....e_b \qquad \Gamma' : e_b[x_i/y_i \cdots] \Downarrow_{\theta_b} \Gamma'' : v}{\Gamma : e\ x_1\ x_2 \cdots \Downarrow_{\mathcal{A}+(1+n+\mathrm{lv}(e))\times\mathcal{S}+\theta_f+\theta_b} \Gamma'' : v}$$

If any of the nested rule matches fail, the rule itself fails. If we assume that they succeed, we can easily obtain $e_b \equiv_f e''_b$. However, the important case here is that we might have $f(x_i) \neq \bot$ due to x_i matching one of the y_i variables from the removed `let` expression. What would happen is that the variable would be propagated into the nested rule match expression. Fortunately, this process establishes $e_b[x_i/y_i] \equiv_f e''_b[x''_i/y''_i]$, so we know that our induction hypothesis holds for the second nested rule match as well. From this it follows directly that the assumption is true for the application rule match as well.

Let us continue with `let` expressions:

$$\frac{\Gamma_a, y_i \mapsto e_i[y_i/x_i, ...], ... : e[y_i/x_i, ...] \Downarrow_\theta \Gamma_b : v}{\Gamma_a : \texttt{let } \{x_i = e_i, ...\} \texttt{ in } e \Downarrow_{\sum(\mathcal{S}+\mathcal{O}(x_i=e_i))+\theta} \Gamma_b : v}$$

As usual equivalence ensures that W'' will encounter a `let` expression in the same position. As far as execution goes, there are two cases: Either we attempt to execute this `let` expression as normal in W'', or this rule application happens to be the one that we want to miraculously ignore in order to study its effects. Let us tackle the latter case first. As mentioned before we chose f so it satisfies $f(y_i) = x_i$, which means that we can directly conclude that $e[y_i/x_i] \equiv_f e''$. As we also defined heap equivalence to ignore the y_i cells it also follows that $\Gamma_a, y_i \mapsto e_i[y_i/x_i, ...], ... \equiv_f \Gamma''_a$ [8]. This means by induction that we also have $\Gamma_b \equiv_f \Gamma''_b$ and $v \equiv_f v''$, which is all we need to prove here.

In the case that this is not the location of our miracle, we know due to variable normalisation (see Section 3.5.1) that our x_i variable names cannot match any of variables bound by of the removed `let` expression, which means that we will find our x_i variables in the same positions in e and e'' as well as e_i and e''_i respectively. If we again assume without loss of generality that we chose the same fresh variables y_i in W and W'' it easily follows that $e[y_i/x_i, ...] \equiv_f e''[y_i/x_i, ...]$ and $e_i[y_i/x_i, ...] \equiv_f e''_i[y_i/x_i, ...]$. This allows us to derive from the induction hypothesis that again we either propagate a failed rule match, or we get $\Gamma_b \equiv_f \Gamma''_b$ and $v \equiv_f v''$.

[8]Note that we actually know that $e = e''$ and $\Gamma_a = \Gamma''_a$, as prior evaluation must have been unchanged. But we do not need this property at the moment.

We can argue very similarly for the `case` expression:

$$\frac{\Gamma_a : e \Downarrow_{\theta_2} \Gamma_b : C_i\ y_j ... \qquad \Gamma_b : e_i[y_j/x_j, ...] \Downarrow_{\theta_3} \Gamma_c : v}{\Gamma_a : \texttt{case}\ e\ \texttt{of}\ \{C_i\ x_j ... \to e_i, ...\} \Downarrow_{\mathcal{E}+n\times\mathcal{S}+\theta_2+\theta_3} \Gamma_c : v}$$

Again a nested rule match fail would cause us to propagate the failure. Thus the scrutinee must evaluate to equivalent values $C_i\ y_j ... \equiv_f C_i\ y''_j ...$, from which we gather that $y_j \equiv_f y''_j$ and therefore $e_i[y_j/x_j, ...] \equiv_f e''_i[y''_j/x_j, ...]$. As we also know $\Gamma_b \equiv_f \Gamma''_b$ we can easily obtain $v \equiv_f v''$ as well as $\Gamma_c \equiv_f \Gamma''_c$. Note that we will have to update this reasoning later due to changed failure propagation in Section 4.6.6.

So far we have simply propagated the renaming and never actually accessed the heap, which is why evaluation proceeded as normal. This makes it quite plausible that we will have more events in W'' than in W'. However, we are still violating program consistency in a major way, which shows for the variable evaluation rules:

$$\Gamma, x \mapsto v : x \Downarrow_{\mathcal{V}+\mathcal{S}} \Gamma, x \mapsto v : v \qquad \frac{\Gamma_a : e \Downarrow_{\theta} \Gamma_b : v}{\Gamma_a, x \mapsto e : x \Downarrow_{\mathcal{V}+(3+\mathrm{lv}(e))\times\mathcal{S}+\mathcal{U}+\theta} \Gamma_b, x \mapsto v : v}$$

For both rule we have two cases: Either we are trying to access a variable with $f(x) = \perp$, which means that $x = x''$ and therefore evaluation in W'' either yields $v \equiv_f v''$ as expected or crashes due to the a failed rule match in the nested evaluation.

On the other hand, where $f(x) \neq \perp$ we must have $x'' = f(x) \neq x$ due to the fact that we chose the variable name x fresh in W when evaluating the removed `let` expression, and especially never introduced it into W''. We also know that there can be no binding for $f(x)$ on the heap Γ''_a. This is because even if we (say) had re-evaluated the `let` expression that we skipped the first time, it would have chosen a fresh variable name instead of $f(x)$, as $f(x)$ here is just the original variable name from the program text. This means that neither (Var1) nor (Var2) can match in W'', so we have to assume a mismatch judgement. Therefore our program crashes in W'' as well:

$$\Gamma' : \perp \Downarrow_{\perp} \Gamma'' : \perp$$

4.6.2 Floating Annotations

The last section has shown that compared to the immediate crash in alternate world W', the program can plausibly continue to run in W''. Not only that, but we actually know that the evaluation in W'' will be equivalent to that in W exactly up to the point where we try to access one of the variables that we “forgot” to push the bindings for. From this it follows that we will have equivalent events in the alternate world as well. In the end, this means that we have a number of events that happen after the match on the `let` expression *without* causally depending on it!

This is the first evaluation rule where we could derive this property, a direct result of us always comparing against an alternate world W' with a total program crash. This especially means that we cannot just use the same annotation strategy as before. After all, now the effects are not isolated "structurally" anymore: As heap references can escape, access to a given heap cell might happen at an arbitrary point of program execution. As this is where our effect is, we will have to find a way to introduce annotations at exactly these points.

Fortunately, visiting all usage sites of a variable is exactly what the semantics by Sestoft [1997] already do, even though its original purpose is to rename variables. Now we can re-use this mechanism for our benefit and put annotations on all variables that we know would cause history to diverge in W'':

$$\frac{\Gamma_a, y_i \mapsto \underline{e}_i[{}_{\langle\alpha\gamma\rangle}y_i/{}_{\langle\gamma\rangle}x_i, ...], ... : \underline{e}[{}_{\langle\alpha\gamma\rangle}y_i/{}_{\langle\gamma\rangle}x_i, ...] \Downarrow_{\theta} \Gamma_b : \underline{v}}{\Gamma_a : {}_{\langle\alpha\rangle}\texttt{let}\ \{x_i = \underline{e}_i, ...\}\ \texttt{in}\ \underline{e} \Downarrow_{\sum\left({}_{\langle\alpha\rangle}\mathcal{S} + {}_{\langle\alpha\rangle}\mathcal{O}(x_i = e_i)\right) + \theta} \Gamma_b : \underline{v}}$$

Similar to how we treated substitution in Section 4.5.2, the notation $\underline{e}[{}_{\langle\alpha\gamma\rangle}y/{}_{\langle\gamma\rangle}x]$ means that we replace the variable while adding the annotation α. This captures the basic fact that in order for the variable expression to evaluate correctly it needs to both exist as well as have an associated binding. Note especially that the substitution is not limited to matching variable expressions, but will also add annotations when substituting variables in constructor or function applications, from where they will get propagated to the actual variable usage sites where appropriate.

Finally, we have to recognise that there is actually another notable difference between W and W''. Namely, the original world has to pay the cost for allocating heap and stack space for the new bindings, which the alternate world W'' skips. This makes these cost events effects of the `let` expression even for our refined alternate world thought experiment, which is why they were annotated with α above. In the end, the jump to W' has still provided us with a much tighter causality tracking than we had before.

4.6.3 Case Expressions

Can we repeat this trick with other rules? The only rule from the cost semantics in Figure 3.4 left to cover is the (Case) rule:

$$\frac{\Gamma_a : e \Downarrow_{\theta_2} \Gamma_b : C_i\ y_j ... \qquad \Gamma_b : e_i[y_j/x_j, ...] \Downarrow_{\theta_3} \Gamma_c : v}{\Gamma_a : \texttt{case}\ e\ \texttt{of}\ \{C_i\ x_j ... \to e_i, ...\} \Downarrow_{\mathcal{E} + n \times \mathcal{S} + \theta_2 + \theta_3} \Gamma_c : v}$$

The straight-forward approach here would be to compare with alternate worlds where the program crashes either because of a mismatch on the `case` expression or because we did not get a constructor back from evaluating the scrutinee. After all, we have already learnt from our discussion of the application rule in Section 4.5.1 how do deal with such interruptions in the middle of a rule application. Selecting all events that come after the mismatch for annotation, we obtain the general annotated `case` evaluation rule:

$$\frac{\Gamma_a : \underline{e} \Downarrow_{\theta_2} \Gamma_b : {}_{\langle\beta\rangle}C_i\ \underline{y}_j ... \qquad \Gamma_b : \underline{e}_i[\underline{y}_j/\underline{x}_j, ...] \Downarrow_{\theta_3} \Gamma_c : \underline{v}}{\Gamma_a : {}_{\langle\alpha\rangle}\texttt{case}\ \underline{e}\ \texttt{of}\ \{C_i\ x_j ... \to \underline{e}_i, ...\} \Downarrow_{\langle\alpha\rangle\mathcal{E}+\langle\alpha\rangle\theta_2+n_i\times\langle\alpha\beta\rangle\mathcal{S}+\langle\alpha\beta\rangle\theta_3} \Gamma_c : {}_{\langle\alpha\beta\rangle}\underline{v}}$$

Just as with the `let` expression, this conservative annotation strategy is valid – but not always optimal.

4.6.4 One-Branch Case

Let us attempt to improve the accuracy of our causality analysis. Ideally, we would like to again find a world W'' that is *closer* to the original world W in that we have a closer match between events. However, this is not quite as obvious here. After all, one important factor in our treatment of `let` was that we knew that its `let` body would get evaluated no matter what. We can not generally do the same for `case` expressions, as we might have a number of branches that evaluation could possibly follow[9].

But what about `case` expression that only have one alternative? Consider a rule specialised to this scenario:

$$\frac{\Gamma_a : e \Downarrow_{\theta_2} \Gamma_b : C_1\ y_j ... \qquad \Gamma_b : e_1[y_j/x_j, ...] \Downarrow_{\theta_3} \Gamma_c : v}{\Gamma_a : \texttt{case}\ e\ \texttt{of}\ \{C_1\ x_j ... \to e_1\} \Downarrow_{\mathcal{E}+n\times\mathcal{S}+\theta_2+\theta_3} \Gamma_c : v}$$

Here we have picked out the situation where the control flow is the most predictable: That scrutinee evaluation produces the expected constructor is enough to guarantee that the highlighted evaluation will occur. And as we are reasoning about the original world, in which we assume that we have a terminating program, this is actually an assumption that we are allowed to make. Note that this applies even if C is not the only constructor of the type in question, as the compiler guarantees that branches cover all possible values. Suppose the following Core code:

```
let {x = ...} in
case x of {C → ...
                 D y → case x of {D y₂ → e₁}}
```

The inner `case` expression has only one branch precisely because the compiler could derive that evaluation can only yield a D constructor. Therefore as long as the scrutinee evaluation terminates, we positively know that e_1 is going to get evaluated.

[9]It would be thinkable however to elect a "default" branch to follow in the alternate world. This would for example allow us to reason about a `case` expressions branching out specific cases such as in Listing 2.5 on page 17. However, it is hard to identify such an intention, see Section 4.7.3 on page 97.

Consequently it actually makes sense to treat e_1 as the alternate-world replacement for the `case` expression in W''. We would attempt the following judgement:

$$\Gamma_a : e_1 \Downarrow_{\theta_3''} \Gamma_c'' : v''$$

Just like with the `let` rule, we will attempt to match rules while leaving x_i bindings in e_1 undefined. Therefore we might get a crash at some point down the road. However as we have seen in Section 4.6.2, this does not need to scare us, as we can easily find and annotate these effects using variable substitution.

On the other hand, we are skipping more code this time around: After all, the rule would normally also evaluate the scrutinee. This not only means that the alternate world W'' loses the cost θ_2, but also all heap changes between Γ_a and Γ_b. Fortunately, the former effect can easily be tracked by adding annotations on θ_2, and the latter can only introduce extra costs in W'' due to extra thunk evaluations. As we are looking for positive causation, the non-existence of costs in W is not something we need to track[10].

4.6.5 Skipping Scrutinisation

So far we have only looked at the effects of matching the one-branch `case` expression. However as explained back in Section 4.5.1, having two nested sub-matches means that we need to investigate the possibility that the rule might mismatch half-way through. To be concrete, we have to account for the alternate-world scenario where we encounter the `case` expression intact, but scrutinee evaluation miraculously does not match the expected $C_1\ y_j ...$ pattern. As before this means that we need to introduce an artificial rule that describes how we suspect the program would react to having the scrutinee result get miraculously replaced by $\bot$. If we just kept the events that temporal order forces, this would mean we get the following rule:

$$\frac{\Gamma_a : e \Downarrow_{\theta_2} \Gamma_b : \bot}{\Gamma_a : \texttt{case}\ e\ \texttt{of}\ \{C_1\ x_j ... \to e_1\} \Downarrow_{\mathcal{E}+n\times\mathcal{S}+\theta_2+\bot} \Gamma_b : \bot}$$

However, this is not satisfactory to us. After all, this means that we would again skip the evaluation of the branch code. Consequently, we would have to annotate branch evaluation results with the cause for evaluation of the scrutinee. Obviously evaluation of the scrutinee will depend on the existence of the `case` expression. Therefore by transitivity the `case` expression causes the branch evaluation, which is exactly what we wanted to disprove in the last section!

[10]Note that we could alternatively prove that these thunk evaluations "move" from the evaluation of the scrutinee to the evaluation of the branch body, similar to what we did in Section 4.4.4. However, this would mean that we would not want to annotate these thunk costs inside θ_2. This would be impossible to notate, so here it actually turns out to be more useful to think of the cost as "reappearing" instead of "moving". We will be seeing quite a bit of this distinction once we get to optimisations.

To fix this situation, we need an artificial “crash recovery” rule like follows:

$$\frac{\Gamma_a : e \ \Downarrow_{\theta_2} \ \Gamma_b : \bot \qquad \Gamma_b : e_1 \ \Downarrow_{\theta_3} \ \Gamma_c : v}{\Gamma_a : \texttt{case}\ e\ \texttt{of}\ \{C_1\ x_j ... \rightarrow e_1\} \ \Downarrow_{\mathcal{E}+n\times\mathcal{S}+\theta_2+\theta_3} \ \Gamma_c : v}$$

So we essentially react to the miraculous scrutinee result mismatch simply by falling back to the same scenario we had before: The evaluation of the naked e_1 branch. In order to uphold temporal consistency as introduced in Section 4.2 on page 59 we make sure that all previous events still happen, and especially pass on the cost θ_2 originating from the “blocked” scrutinee evaluation.

If we now put all the pieces together, we can obtain the result we wanted: No matter whether our miracle is the `case` expression disappearing or the scrutinee result mismatching, the alternate world will still end up evaluating the branch until the first usage site of a bound variable. Therefore all branch costs and results up to that point are caused by neither. From this realisation we obtain an annotated one-branch `case` rule:

$$\frac{\Gamma_a : \underline{e} \ \Downarrow_{\theta_2} \ \Gamma_b : {}_{\langle\beta\rangle}C_1\ \underline{y}_j ... \qquad \Gamma_b : \underline{e}_1[{}_{\langle\alpha\beta\gamma\rangle}\underline{y}_j / {}_{\langle\gamma\rangle}x_j, ...] \ \Downarrow_{\theta_3} \ \Gamma_c : \underline{v}}{\Gamma_a : {}_{\langle\alpha\rangle}\texttt{case}\ \underline{e}\ \texttt{of}\ \{C_1\ x_j ... \rightarrow \underline{e}_1\} \ \Downarrow_{{}_{\langle\alpha\rangle}\mathcal{E}+n\times{}_{\langle\alpha\rangle}\mathcal{S}+{}_{\langle\alpha\rangle}\theta_2+\theta_3} \ \Gamma_c : \underline{v}}$$

Note that in comparison to the previous formulation the $\alpha\beta$ annotation has now moved from θ_3 and $\underline{v}$ respectively to $\underline{y}_i$, analogous to what happened when we discussed the `let` rule in Section 4.6.2 on page 88.

4.6.6 Crash Recovery Consistency

While the conclusions might sit well with us, we have to take care that we are not introducing inconsistencies into our causality model. After all, the very structure of the proposed crash recovery rule is worrisome: We are matching on a $\bot$ result in order to do a second nested rule match. This violates the assumption that a failed child rule match will always cause the parent rule match to fail. From the point of view of the event model, our intention is clearly to have certain events to happen independently from others, but if we take the rule at face value we are actually proposing for events to happen as a reaction to other events *not* happening.

As a result we should critically review our argumentation up to this point for inconsistencies. Fortunately, we know that the artificial rule will only ever match in the alternate world, as the original world will never see rule mismatches. Unfortunately though, we actually have encountered one scenario where we want to pass a $\bot$ scrutinee evaluation result “through”. Specifically, when we were reasoning about the effects of “forgetting” bindings in Section 4.6.1 we assumed that encountering an undefined binding would cause all further events to vanish. However, with our addition this is now wrong – in the alternate world the program can now “come back from the dead”.

To understand how, consider the following Core fragment:

$$\begin{aligned}&\texttt{let}\,\{x = ...\}\,\texttt{in}\\ &\texttt{case}\ x\ \texttt{of}\,\{D\ y \to e\}\end{aligned}$$

Here the crash recovery rule will actually change the outcome of removing the x binding. After all, we would look for its effects by considering the following alternate-world scenario:

$$\texttt{case}\ \bot\ \texttt{of}\,\{D\ y \to e\}$$

Normally we would expect the evaluation of this to fail, which yields us to the conclusion that the evaluation result of the program depends causally on x. However, with our crash recovery rule, this is not true any more: In the alternate world we would continue evaluating e, which might well end up producing the same result we had in the original world. Therefore the result now would *not* be an effect of x any more!

This change in interpretation means that we need to adapt our reasoning from Section 4.6.1 on page 86. Back then we assumed that every rule application could either crash in the alternate world, or produce equivalent results. Now we have to acknowledge the fact that at some point we could recover from a crash in W'':

$$\frac{\Gamma''_a : e'' \Downarrow_{\theta''_2} \Gamma''_b : \bot \qquad \Gamma''_b : e''_1 \Downarrow_{\theta''_3} \Gamma''_c : v''}{\Gamma''_a : \texttt{case}\ e''\ \texttt{of}\ \{C_1\ x_j ... \to e''_1\} \Downarrow_{\mathcal{E}+n\times\mathcal{S}+\theta''_2+\theta''_3} \Gamma''_c : v''}$$

while naturally encountering a normally evaluated `case` expression in W:

$$\frac{\Gamma_a : e \Downarrow_{\theta_2} \Gamma_b : C_1\ y_j ... \qquad \Gamma_b : e_1[y_j/x_j, ...] \Downarrow_{\theta_3} \Gamma_c : v}{\Gamma_a : \texttt{case}\ e\ \texttt{of}\ \{C_1\ x_j ... \to e_1\} \Downarrow_{\mathcal{E}+n\times\mathcal{S}+\theta_2+\theta_3} \Gamma_c : v}$$

Due to our induction hypothesis we know that $\Gamma_a \equiv_f \Gamma''_a$ and $e \equiv_f e''$. We can even arrange $e_1[y_j/x_j, ...] \equiv_g e''_1$ by deriving a new mapping function g from f with $g(y_j) = x_j$. However, note that we have no direct way for establishing $\Gamma_b \equiv_f \Gamma''_b$, as we have to assume that while in "limbo" W'' might have skipped thunk updates that happened in Γ_b. However, we can fix the proof by introducing a mapping from unevaluated bindings in W'' to their evaluation in W, allowing us to argue that delayed evaluation in W'' yields the same results as in W. Therefore we have again at worst moved extra cost to a later point in W'', which we do not need to explicitly track. We have shown before how such a proof would work, and will skip the full explanation at this point.

In the end, once we have ironed out the inconsistency the obtained result is basically the same as in Section 4.6.1: The concrete "surviving" event set has changed, but the world W'' is still closer to W than W'. Furthermore, we can show that the new rules together propagate annotations exactly to the events that differ between W and W''.

$$\Gamma : {}_{\langle\alpha\rangle}C\ \underline{x}_1\ \underline{x}_2 ... \Downarrow_{\langle\alpha\rangle\mathcal{C}+n\times_{\langle\alpha\rangle}\mathcal{H}} \Gamma : {}_{\langle\alpha\rangle}C\ \underline{x}_1\ \underline{x}_2 ... \tag{Con}$$

$$\Gamma : {}_{\langle\alpha\rangle}\lambda y.\underline{e} \Downarrow_{\langle\alpha\rangle\mathcal{L}+\mathrm{lv}(e)\times_{\langle\alpha\rangle}\mathcal{H}} \Gamma : {}_{\langle\alpha\rangle}\lambda y.\underline{e} \tag{Lam}$$

$$\frac{\Gamma : \underline{e} \Downarrow_{\theta_f} \Gamma' : {}_{\langle\beta_1\rangle}\lambda y_1 . {}_{\langle\beta_2\rangle}\lambda y_2 .... \underline{e}_b \qquad \Gamma' : \underline{e}_b[{}_{\langle\gamma\rangle}x_i / {}_{\langle\gamma\rangle}y_i \cdots] \Downarrow_{\theta_b} \Gamma'' : \underline{v}}{\Gamma : {}_{\langle\alpha\rangle}(\underline{e}\ \underline{x}_1\ \underline{x}_2 \cdots) \Downarrow_{\langle\alpha\rangle\mathcal{A}+(1+n+\mathrm{lv}(e))\times_{\langle\alpha\rangle}\mathcal{S}+_{\langle\alpha\rangle}\theta_f+_{\langle\alpha\overline{\beta_i}\rangle}\theta_b} \Gamma'' : {}_{\langle\alpha\overline{\beta_i}\rangle}\underline{v}} \tag{App}$$

$$\frac{\Gamma_a, y_i \mapsto \underline{e}_i[{}_{\langle\alpha\gamma\rangle}y_i / {}_{\langle\gamma\rangle}x_i, ...], ... : \underline{e}[{}_{\langle\alpha\gamma\rangle}\underline{y}_i / {}_{\langle\gamma\rangle}x_i, ...] \Downarrow_{\theta} \Gamma_b : \underline{v}}{\Gamma_a : {}_{\langle\alpha\rangle}\texttt{let}\ \{x_i = \underline{e}_i, ...\}\ \texttt{in}\ \underline{e} \Downarrow_{\sum({}_{\langle\alpha\rangle}\mathcal{S}+_{\langle\alpha\rangle}\mathcal{O}(x_i=e_i))+\theta} \Gamma_b : \underline{v}} \tag{Let}$$

$$\Gamma, x \mapsto \underline{v} : {}_{\langle\alpha\rangle}x \Downarrow_{\langle\alpha\rangle\mathcal{V}+_{\langle\alpha\rangle}\mathcal{S}} \Gamma, x \mapsto \underline{v} : {}_{\langle\alpha\rangle}\underline{v} \tag{Var1}$$

$$\frac{\Gamma : \underline{e} \Downarrow_{\theta} \Gamma' : \underline{v}}{\Gamma, x \mapsto \underline{e} : {}_{\langle\beta\rangle}x \Downarrow_{\langle\beta\rangle\mathcal{V}+(3+\mathrm{lv}(e))\times_{\langle\beta\rangle}\mathcal{S}+_{\langle\beta\rangle}\mathcal{U}+_{\langle\beta\rangle}\theta} \Gamma', x \mapsto \underline{v} : {}_{\langle\beta\rangle}\underline{v}} \tag{Var2}$$

$$\frac{\Gamma_a : \underline{e} \Downarrow_{\theta_2} \Gamma_b : {}_{\langle\beta\rangle}C_1\ \underline{y}_j ... \qquad \Gamma_b : \underline{e}_1[{}_{\langle\alpha\beta\gamma\rangle}\underline{y}_j / {}_{\langle\gamma\rangle}x_j, ...] \Downarrow_{\theta_3} \Gamma_c : \underline{v}}{\Gamma_a : {}_{\langle\alpha\rangle}\texttt{case}\ \underline{e}\ \texttt{of}\ \{C_1\ x_j ... \to \underline{e}_1\} \Downarrow_{\langle\alpha\rangle\mathcal{E}+n\times_{\langle\alpha\rangle}\mathcal{S}+_{\langle\alpha\rangle}\theta_2+\theta_3} \Gamma_c : \underline{v}} \tag{Case1}$$

$$\frac{\Gamma_a : \underline{e} \Downarrow_{\theta_2} \Gamma_b : {}_{\langle\beta\rangle}C_i\ \underline{y}_j ... \qquad \Gamma_b : \underline{e}_i[{}_{\langle\gamma\rangle}\underline{y}_j / {}_{\langle\gamma\rangle}x_j, ...] \Downarrow_{\theta_3} \Gamma_c : \underline{v}}{\Gamma_a : {}_{\langle\alpha\rangle}\texttt{case}\ \underline{e}\ \texttt{of}\ \{C_i\ x_j ... \to \underline{e}_i, ...\} \Downarrow_{\langle\alpha\rangle\mathcal{E}+_{\langle\alpha\rangle}\theta_2+n_i\times_{\langle\alpha\beta\rangle}\mathcal{S}+_{\langle\alpha\beta\rangle}\theta_3} \Gamma_c : {}_{\langle\alpha\beta\rangle}\underline{v}} \tag{CaseN}$$

Figure 4.19: Causality Model

4.7 Causality Model

At this point we have run through all rules of our performance model, with the resulting causality model rules collected in Figure 4.19. We have added a good number of of extensions and annotations to Launchbury's original semantics, which might make the full semantics a bit hard to follow. Yet when we look back, the analysis actually boiled down just a few basic principles:

- Every possible point where a rule could mismatch means that we need to put their annotation on terms that are produced "after" them.
- The only exceptions are cases where we can identify a better alternate world to compare with. This has generally to do with bindings, the effects of which we track by attaching annotations to the variables.
- Heaps can get passed through freely, and in particular we can update thunks without annotating the cause for their evaluation.

This should make it easier to remember later on where exactly we expect annotations to go. However, before we starting putting this model to work we still have a few loose ends to tie up, which we will do in this section.

4.7.1 Annotation Encapsulation

If we look at the list of principles, the last one should stick out: We proved back in Section 4.4 on page 72 that we can update thunks freely, but obviously this is only one type of change that we could observe on the heap. Let us shed some more light on this. Suppose that we have an arbitrary rule that gives us a judgement:

$$\Gamma_1 : e \Downarrow_\theta \Gamma_2 : v$$

to find its effects we have at various occasions used "crash" alternative judgements [11]. This generally looked like follows:

$$\Gamma_1 : e \Downarrow_\perp \Gamma_1 : \perp$$

Finding effects is now all about comparing the alternate world with the original, and as the heap stays the same in W', this comes down to comparing Γ_1 with Γ_2. But what exactly constitutes a change here? Clearly heap cells that have the same value do not count. And as we have showed in Section 4.4, replacing a thunk by its value is no "material" change either, as the alternative would be simply re-computing it later.

Given the way that our operational semantics work, this actually covers all but one case: What happens if a (Let) rule introduces a new binding on Γ_2 that did not exist on Γ_1? Clearly this is no immaterial effect, as "forgetting" the binding might provoke a program crash once the variable gets evaluated. However, now that we have finished our causality semantics, we have actually new tools at our disposal: We argue that all possible effects will *already* receive the appropriate annotation from other channels.

Specifically, note that we know that x must have been introduced by a (Let) rule, so it must be a fresh name. For our rule match, this means that the rest of the program can only access the new heap cell if the return value v has a reference to it – either directly or indirectly. So suppose an alternate world where our rule miraculously does not crash, but returns $\perp$ anyway. Then we know that the rest of the program has no way of accessing x, even if it still exists on the heap. Therefore the effects of the x heap cell are strictly fewer than that of returning v, which means that as long as we are consistent with tracking value effects, annotating v with the cause in question is actually enough [12]. And as we can see in Figure 4.19, all of our rules generally add their annotation on the result value, therefore "encapsulating" the heap reference. The only two rules where this is not the case are `let` and one-branch `case`, where the different alternate world does not require us to annotate in the first place.

[11] This generalises to "intermediate" crashes as explained in Section 4.5.1 on page 82 as well.

[12] It might seem dangerous to rely on consistency while trying to establish consistency – yet in fact either viewpoint is internally consistent, so we just choose the more intuitive one.

Note that this situation bears some familiarities to the nested events we introduced in Section 4.3.4: There we remarked that nested events cannot normally "escape" their parent event either. However, we have actually already broken this particular principle: In Section 4.6 we took sub-expressions of `let` and `case` expressions respectively and evaluated them without causal dependency on the parent expression. In the event model, this means that we actually "looked through" the annotation on the parent expression. Fortunately, there was never any reason to do this for values, or we might have a harder time arguing here. This should demonstrate how reasoning about causality is all about internal consistency.

4.7.2 Close Causes

Up to this point we have seen two example in Section 4.6.2 and Section 4.6.4 where we could make our causality analysis more exact by comparing against a tailor-made alternate world. A hypothetical optimal closest world is firmly outside of our grasp, as it would essentially solve the performance problem for us. But it still begs the question: Are there other – closer – worlds out there that we could use to further enhance the expressiveness of our model?

Let us remind ourselves of the rule match "miracles" that we have used so far. Keeping in mind the two mentioned exceptions, we use the original-world expression e to chose the alternate-world expression e' as follows:

$$e' = \begin{cases} e_1 & \text{where} \quad e = \texttt{let}\ \{...\}\ \texttt{in}\ e_1 \\ e_1 & \text{where} \quad e = \texttt{case}\ ...\ \texttt{of}\ \{... \to e_1\}\ \text{ with } e_1 \text{ sole branch} \\ \bot & \text{otherwise} \end{cases}$$

It makes sense to limit ourselves to such replacement rules: After all, we ideally want small miracles, and switching out a term by another is the kind of change that disturbs the consistency of the system the least.

What makes a good replacement for the purpose of our operational semantics? As explained in Section 4.3.3 on page 67 every time we reference a term with nested annotations this implies a whole tree of abstract events, each corresponding to a certain part of the expression having a certain form. If we want to study effects of the cause annotated on the top-level, this means that our primary task is eliminating the top-level event while maintaining consistency. If we find that we can do this without eliminating nested events, this means that we can view them as "independent" for the purpose of the considered causal process. For example, when we proposed the new alternate world for the (Let) rule we essentially thought of the `let` binding as an "attachment" to its otherwise independent body.

To summarise, we are looking for replacement rules that keep as many nested events (and therefore sub-terms) as possible while having implications that match the original world as closely as possible. However, we must still be careful. Remember from our introduction to causality theory in Section 2.3 that we might end up swapping a tree for a wall: We have to be wary of replacing equals with equals. We could for example propose the following replacement:

$$e' = \texttt{let}\,\{x = C\ y\}\,\texttt{in}\ x \quad \text{where} \quad e = C\ y$$

This "replacement" introduces the very same functionality, just using a different mechanism. This is clearly silly, and would be completely useless for causality analysis. After all, both versions would allocate the constructor, possibly leading us to the conclusion that the associated cost was not actually caused by the constructor! Making such mistakes could lead to subtle errors in our reasoning, so we should be careful in making sure that our replacement unambiguously encodes a "removal".

From the point of view of the event model this means that after eliminating all "cause" events we would like to avoid fabricating any extra events in order to restore some notion of causality. Instead we should make do with re-interpreting existing events in new roles. As every annotated sub-term corresponds to an event in our interpretation, this means that the conservative choice here is therefore to restrict ourselves exactly to proper annotated sub-terms of the original.

4.7.3 Close Effects

However, we have to not only consider the miracle itself when deciding on the closest world to chose. After all, what made this viewpoint work for `let` and `case` expression was not only that removing the top-level expression in this way required relatively few "miracle" changes, but also that it changed future alternate world events in a controlled fashion: We were able to predict accurately which events would persist and which ones we would expect to vanish. More critically, we could derive that most of these events had equivalents in the original execution, and could find a way to encode this fact as annotations for our causality semantics.

Specifically, in both instances we found that we could predict that executing the original-world expression e would result in a sub-term e' getting evaluated in a closely related context. This made it plausible that we could simply evaluate "past" the rule application in the alternate world. Even more critically, we could easily describe at what point evaluation would deviate: Variables without associated heap bindings provided nice hooks for us to bind annotations to.

Note however that even though we limited ourselves to proper sub-expression replacements in the last section, most possible choices will not behave as nicely: For example, replacing an expression with a replacement term of a different type is bound to have unpredictable consequences, as we are violating the assumptions that went into compilation. For example, the following replacement rule would simply be nonsensical:

$$e' = f \quad \text{where} \quad e = f\ x$$

This would allow us to see evaluation of the function expression as happening independent of the function application. However in contrast to `let` bindings, viewing an application as an "attachment" does not make much sense in the context of the larger program: The evaluation result will have a higher arity than the rest of the program expects. This might simply lead to a crash at the next value usage site, but in the worst case it could actually end up leading to a program evaluation that has nothing to do with the original. This would not only be difficult to describe in the form of an annotated evaluation rule, but is also unlikely to gain us any insight.

So at this point the only replacements we are willing to allow are proper sub-expressions that are guaranteed to evaluate with a result that does not clearly disturb code invariants. This already reduces our options greatly: Looking at the expression types we defined in Section 3.5.2 on page 43 and Section 4.3.3 on page 67 respectively, we observe that indeed only `let` bodies and `case` branches are guaranteed to share the type of their respective parent expressions.

4.7.4 Complexity

This still leaves more deeply nested sub-expressions. Could there be cases where we could find viable expression replacements buried deeper within the tree? In fact, it is not hard to construct examples for this. Consider for instance:

$$e' = e_1[y/x] \quad \text{where} \quad e = (\lambda x.e_1)\ y$$

This rule satisfies all of our requirements, after all e_1 is a proper sub-expression and variable substitution qualifies as simply putting events into a different context in our eyes. Moreover, we know that β-reduction will never change program meaning in any way, so the changes to program evaluation are about as predictable as they can get.

But still – this rule does not provide much value to us. Note that we would have to introduce a rather awkward special case into the application rule to implement this. This technicality is significant, because for considering optimisations later we want annotations to be predictable, and as Section 4.8.2 will show, the special cases introduced thus far already make that tricky. "Anticipating" code transformations like this is something we should only do where it reduces complexity in some way.

```
1  fun = ...
2      where !a = ...
3            b = ...
```

(Haskell)

```
fun = case ... of {a →
        let {b = ...} in ...
      }
```

(Core)

Figure 4.20: Core Representation of Haskell Bindings

However, let us also briefly consider the possible intuition behind the β-reduction replacement rule. Can we really view applying a lambda expression as a detachable "term fragment"? Note that we are currently considering unoptimised code, so this really comes back to the supposed intention of the programmer: Would we expect them to build such obviously reducible code without a good reason? Would we really help them by declaring costs inside e_1 as independent of the source code corresponding to either the application or the lambda expression? The answer is probably "no", but it is quite hard to pin-point exactly why this is the case.

4.7.5 Intuition

As the last sections highlighted, choosing replacement terms for closest-world thought experiments is not only about technical requirements, but also about the programmer's intuitive view of the program. This begs the question: How do the special cases for `let` and one-branch `case` expressions fare from this particular angle? There is no way we can answer this conclusively, as it would be quite hard to come up with a way to measure intuitiveness. However, we can observe anecdotal evidence simply by looking at the Haskell syntax. After all, computer languages are built specifically to make it easy to express common patterns that come up during programming, so we can expect such patterns to correspond closely to the intuition of typical Haskell program authors.

For instance, consider the program fragment from Figure 4.20: Here we bind two variables `a` and `b`, and make use of the `BangPatterns` language extension in order to request strict evaluation of the `b` binding. If we look at the same code after desugaring and preparation (see Section 3.5.1), we observe that lazy and strict `where` bindings correspond directly to `let` and one-branch `case` expressions respectively. Not accidentally, these happen to be exactly the expressions that we have defined specialised closest-world rules for. Therefore there actually is a strong intuition behind this choice: Both correspond directly to a Haskell binding, therefore we can easily interpret our alternate world thought experiments as simply "commenting out" the line 2 or 3 from the Haskell program in Figure 4.20. This is probably what a Haskell programmer would have done if we had asked them to naïvely "remove" the bindings.

4.8 Optimisations

At this point we have finished our causality analysis of the performance model from Figure 3.4 on page 55. Our price was on display in Figure 4.19 on page 94: Every rule is now able to accept and produce fully annotated terms. This means that conditional on input causes and our analysis being correct, we now have a systematic way of deriving a correct-by-construction cost mapping θ: A program profile. As we have derived the performance model from an approximation of Haskell program performance behaviour, it is furthermore plausible that we have gained a good idea of the causal dependencies involved in executing an unoptimised program.

However, this means that we still have not quite reached our goal. After all, real Haskell programs generally go through a number of optimisation passes before they actually get executed. This means that the causal connections that we have identified will get changed and obfuscated by code transformations rewriting the very source code program evaluation takes as input. However, this does not mean that our work has been useless: After all, the whole point of optimisations is that they know enough about the mechanisms of evaluation to arrange expressions so that the program arrives at exactly the same result – just using fewer resources. We will now do the same for cause annotations: Arrange for expression annotations to retain a sensible profiling view of all remaining costs as well as possibly introduced overheads. From a practical point of view, limiting ourselves to only changing annotations means that the involvement of optimisations does not increase the complexity of generating profiling data in the first place. This is a very desirable property.

To get there, we will again employ Lewis [1973] counter-factual causality reasoning, just as we did in the last section. However this time around we will actually do so in a very specific fashion. Given a code transformation of the form

$$e' \quad \Longrightarrow \quad e$$

our original world W will always be the evaluation of the optimised program e whereas the original evaluation of e' will act as our point of comparison W'.

Furthermore, we know that the effects we are looking for are the ones that can be observed from the "outside". For the purpose of our performance model, this excludes the details about all the events happening during evaluation, but includes values and costs produced by the top-level judgement. Fortunately, we already know that optimisations will never change the program results, so our goal will simply be to keep their annotations in place wherever we can. On the other hand, the whole point of optimisations is to influence the performance of the program, therefore we will have to take more care with the conclusion we make about the causal history behind costs.

4.8.1 Beta Reduction

To get started, consider the situation where we apply a function term which happens to be a lambda expressions. Such code can be reduced using the well-known β-reduction optimisation. Incorporating annotations, this rule would look like follows:

$$_{\langle\alpha\rangle}\big((_{\langle\beta\rangle}\lambda y.\underline{e})\ x\big) \quad\Longrightarrow\quad _{\langle ?\rangle}\underline{e}[\underline{x}/\underline{y}]$$

The "?" placeholder reminds us of the fact that we do not yet know what annotations would make sense for the right-hand side. After all, when we considered looking through such constructs in Section 4.7.3, we came to the conclusion that we want to see this type of code as intentionally constructed, so we would not want to lose our capability of following that intention backwards. This means that we need to find a way to re-introduce the causal connections given by α and β into the optimised program.

Let us have a look at how evaluation of the transformed part of the program looked like before and after the optimisation. To do that, we simply plug $_{\langle\alpha\rangle}\big((_{\langle\beta\rangle}\lambda y.\underline{e})\ x\big)$ into the causality model from Figure 4.19, and obtain:

$$\frac{\Gamma_a : {}_{\langle\beta\rangle}\lambda y.\underline{e}\ \Downarrow_{{}_{\langle\beta\rangle}\mathcal{L}+\mathrm{lv}(e)\times{}_{\langle\beta\rangle}\mathcal{H}}\ \Gamma_a : {}_{\langle\beta\rangle}\lambda y.\underline{e} \qquad \Gamma_a : \underline{e}[\underline{x}/\underline{y}]\ \Downarrow_{\theta_b}\ \Gamma_b : \underline{v}}{\Gamma_a : {}_{\langle\alpha\rangle}\big((_{\langle\beta\rangle}\lambda y.\underline{e})\ \underline{x}\big)\ \Downarrow_{{}_{\langle\alpha\rangle}\mathcal{A}+\left(1+n+\mathrm{lv}(e)\right)\times{}_{\langle\alpha\rangle}\mathcal{S}+{}_{\langle\alpha\beta\rangle}\mathcal{L}+\mathrm{lv}(e)\times{}_{\langle\alpha\beta\rangle}\mathcal{H}+{}_{\langle\alpha\beta\rangle}\theta_b}\ \Gamma_b : {}_{\langle\alpha\beta\rangle}\underline{v}}$$

If we use the same heap to evaluate the optimised version $\underline{e}[\underline{x}/\underline{y}]$, we simply get:

$$\Gamma_a : \underline{e}[\underline{x}/\underline{y}]\ \Downarrow_{\theta_b}\ \Gamma_b : \underline{v}$$

Ignoring annotations for a moment, this happens to be how we would prove that β-reduction is valid in the first place. After all, the sub-match in the unoptimised program exactly matches the judgement of the optimised expression, therefore both programs are guaranteed to arrive not only at the same value v but also the same heap Γ_b.

When bringing causal annotations into play things look a bit different, though. After all, the original execution had an extra $\alpha\beta$ annotation on the result value. Furthermore, the cost term actually sees substantial changes. We have:

$$\begin{aligned} \theta &= && \theta_b \\ \theta' &= {}_{\langle\alpha\rangle}\mathcal{A} + \big(1 + n + \mathrm{lv}(e)\big) \times {}_{\langle\alpha\rangle}\mathcal{S} + {}_{\langle\alpha\beta\rangle}\mathcal{L} + \mathrm{lv}(e) \times {}_{\langle\alpha\beta\rangle}\mathcal{H} + {}&& {}_{\langle\alpha\beta\rangle}\theta_b \end{aligned}$$

The comparison shows why β-reduction is an optimisation: We see that a significant cost term simply vanishes – and as we only care about events that actually exist in the real world W, we can ignore them for causality analysis. And yet this still leaves the remaining cost term θ_b, which along with the result value has a different annotation from before.

4.8.2 Push Annotations

In order to formulate a useful β-reduction rule operating on annotated programs, we ideally want to make sure that all surviving costs keep the same annotations as before. As annotations on values can cause annotations on costs to change, this also indirectly applies to value annotations. In our case, this means that for the evaluation of the optimised expression we want an $\alpha\beta$ annotation on both the cost term θ_b as well as the result value v.

Changing causality or evaluation semantics is out of the question, so the only way of achieving this is to suitably re-annotate the expression in the course of program optimisation. Speaking generally, given an expression $\underline{e}$ and an annotation α, we want an expression ${}_{\langle\!\langle\alpha\rangle\!\rangle}\underline{e}$ such that:

$$\Gamma : \underline{e} \Downarrow_\theta \underline{v} \quad\Longrightarrow\quad \Gamma : {}_{\langle\!\langle\alpha\rangle\!\rangle}\underline{e} \Downarrow_{\langle\alpha\rangle\theta} {}_{\langle\alpha\rangle}\underline{v}$$

Depending on what kind of expression $\underline{e}$ happens to be, this might be as simple as setting ${}_{\langle\!\langle\alpha\rangle\!\rangle}\underline{e} = {}_{\langle\alpha\rangle}\underline{e}$. Our causality semantics in Figure 4.19 on page 94 show that this would in fact work for constructor, lambda, application, variable as well as non-one-branch `case` expressions: The rules that consume these expressions share the trait that they add the expression annotation both on all costs as well as on the returned value. Therefore an extra annotation on the expression will end up exactly where we would like to see it.

On the other hand, this still leaves the special cases introduced in Section 4.6.2 and Section 4.6.4 for us to circumnavigate. After all, in these instances we came to the conclusion that the expression fragment in question could easily be removed and therefore produced costs and values were not causally dependent on the expression. Fortunately, in both cases the costs and values that would lack an annotation originated from nested expressions, so we can simply "push" the right annotation inwards until we hit an expression where they will "stick":

$$\begin{array}{rcll} {}_{\langle\!\langle\alpha\rangle\!\rangle}{}_{\langle\beta\rangle}\texttt{let}\,\{...\}\,\texttt{in}\ \underline{e}_1 &=& {}_{\langle\alpha\beta\rangle}\texttt{let}\,\{...\}\,\texttt{in}\ {}_{\langle\!\langle\alpha\rangle\!\rangle}\underline{e}_1 & \\ {}_{\langle\!\langle\alpha\rangle\!\rangle}{}_{\langle\beta\rangle}\texttt{case}\ ...\ \texttt{of}\ \{C\ ... \to \underline{e}_1\} &=& {}_{\langle\alpha\beta\rangle}\texttt{case}\ ...\ \texttt{of}\ \{C\ ... \to {}_{\langle\!\langle\alpha\rangle\!\rangle}\underline{e}_1\} & \\ {}_{\langle\!\langle\alpha\rangle\!\rangle}\underline{e} &=& {}_{\langle\alpha\rangle}\underline{e} & \text{otherwise} \end{array}$$

Note that the structure of this definition is very similar to what we identified as alternate-world replacements in Section 4.7.2, as we are mirroring our earlier choice of alternate world expression. This should illustrate why we advised caution when injecting extra complexity into our causality model in Section 4.7.4 on page 98. Any extra effort at that stage would have to be "undone" at this point!

However, the above definition encapsulates these details nicely, which allows us to state the new causality-aware β-reduction rule in a compact fashion:

$$_{\langle\alpha\rangle}((_{\langle\beta\rangle}\lambda y.\underline{e})\ x) \quad\Longrightarrow\quad _{\langle\!\langle\alpha\beta\rangle\!\rangle}\underline{e}[\underline{x}/\underline{y}]$$

As all cost for evaluating the new expression can be matched to cost generated by the old expression carrying the very same annotations, this rule retains perfect profile accuracy.

4.8.3 Effects on Global Profile

At this point we have understood that the β-reduction rule retains profile accuracy if we consider just the evaluation of a single changed expression in the same context. However, strictly speaking this is not what we set out to do. What we want is a statement about the accuracy of the entire program profile.

First note that we actually do not know the heaps to be equal in both worlds ($\Gamma_a = \Gamma'_a$) due to the fact that we might have unevaluated heap cells mentioning the optimised expression. In order to prove any properties formally we have to talk about some notion of term equivalence, where we define $\underline{e} \equiv \underline{e}'$ in a similar way to Section 4.4.2 on page 73. While back then we allowed variable names to change, this time around we make a special case for the optimised expression:

$$_{\langle\!\langle\alpha\beta\rangle\!\rangle}\underline{e}[\underline{x}/\underline{y}] \equiv {}_{\langle\alpha\rangle}((_{\langle\beta\rangle}\lambda y.\underline{e}')\ x) \quad \text{where } \underline{e} \equiv \underline{e}'$$

We extend this in the natural way to heap equivalence $\Gamma_a \equiv \Gamma'_a$.

Furthermore suppose that a profile comparison function $\leq$ that behaves as expected with respect to profile composition and annotation. For example given arbitrary $\theta_1 \leq \theta'_1$ and $\theta_2 \leq \theta'_2$ we should have $\theta_1 + \theta_2 \leq \theta'_1 + \theta'_2$ as well as $_{\langle\alpha\rangle}\theta_1 \leq {}_{\langle\alpha\rangle}\theta'_1$ for all α. In the case of β-reduction we could simply define this as:

$$\begin{array}{rcll} \emptyset & \leq & \emptyset & \\ \underline{\theta} + \underline{\mathcal{O}} & \leq & \theta' + \underline{\mathcal{O}} & \text{where } \theta \leq \theta' \\ \underline{\theta} & \leq & \theta' + \underline{\mathcal{O}} & \text{where } \theta \leq \theta' \end{array}$$

in order to ensure that our optimisation actually led to a strict reduction in cost. This definition quite obviously satisfies the laws, however note that we will revisit this definition later in Section 4.8.5 where we can not establish as strong a result.

Now let us suppose two judgements in the original and alternate world respectively:

$$\Gamma_a : \underline{e} \Downarrow_\theta \Gamma_b : \underline{v} \quad \text{and} \quad \Gamma'_a : \underline{e}' \Downarrow_{\theta'} \Gamma'_b : \underline{v}'$$

with $\Gamma_a \equiv \Gamma'_a$ and $\underline{e} \equiv \underline{e}'$. Now we can prove inductively that we have $\Gamma_b \equiv \Gamma'_b$, $\underline{v} \equiv \underline{v}'$ as well as $\theta \leq \theta'$. To see this, we note that if $\underline{e}'$ does not happen to be transformed, we can derive the property easily simply by the structure of the rules in Figure 4.19. However, if we find an optimised expression, we can use a variant of the argument from Section 4.8.1 to show that, conditional on all sub-matches adhering to the induction assumption, it also must be true for the evaluation of the expression itself.

In the end, what we get out is the statement that we must have $\theta \leq \theta'$ for the whole program. Given our definition of profile comparison this means that have not only potentially reduced cost, but also more importantly retained all relevant annotations.

4.8.4 Floating Let

The need to define special notation for "pushing" an annotation on costs and values might be surprising. After all, now it might appear like the special cases we introduced for `let` and `case` expressions actually made our job harder! However, the opposite side of the coin is that there are optimisations where this actually makes sensible annotation possible in the first place. Consider for example a simple instance of `let` floating:

$$\begin{aligned}
&{}_{\langle\alpha_c\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \rightarrow {}_{\langle\alpha_l\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \underline{e}_3, ...\} \\
&\qquad\qquad\qquad \Longrightarrow \quad {}_{\langle\beta_l\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \big({}_{\langle\beta_c\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \rightarrow \underline{e}_3, ...\}\big)
\end{aligned}$$

What this does is to take the y binding and move it up through the enclosing `case` expression. While it might not be quite obvious whether this transformation actually improves program performance, this is often an intermediate step to further optimisations, such as turning $\underline{e}_2$ into a global constant [Peyton Jones et al., 1996].

Our goal is now to derive new annotations β_l and β_c given the existing ones. To do this, we first need an idea of how these annotations will actually propagate to the final program profile. Consider the evaluation of the *optimised* expression in the world W, assuming that the $\underline{e}_3$ branch gets taken:

$$\begin{array}{l}
\Gamma : {}_{\langle\beta_l\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \big({}_{\langle\beta_c\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \rightarrow \underline{e}_3, ...\}\big) \\
\left[\begin{array}{l}
\Gamma, \hat{y} \mapsto \underline{e}_2[{}_{\langle\beta_l\gamma\rangle}\hat{y}/{}_{\langle\gamma\rangle}y] : {}_{\langle\beta_c\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \Big\{C\ x \rightarrow \underline{e}_3[{}_{\langle\beta_l\gamma\rangle}\hat{y}/{}_{\langle\gamma\rangle}y], ...\Big\} \\
\left[\begin{array}{l}
\Gamma, \hat{y} \mapsto \underline{e}_2[{}_{\langle\beta_l\gamma\rangle}\hat{y}/{}_{\langle\gamma\rangle}y] : \underline{e}_1 \ \Downarrow_{\theta_1}\ \Gamma_1 : {}_{\langle\delta\rangle}C\ \underline{\hat{x}} \\
\Gamma_1 : \underline{e}_3[{}_{\langle\beta_l\gamma\rangle}\hat{y}/{}_{\langle\gamma\rangle}y][{}_{\langle\gamma\rangle}\underline{\hat{x}}/{}_{\langle\gamma\rangle}x] \ \Downarrow_{\theta_2}\ \Gamma_2 : \underline{v}
\end{array}\right. \\
\Downarrow_{{}_{\langle\beta_c\rangle}\mathcal{E}+{}_{\langle\beta_c\rangle}\theta_1+{}_{\langle\beta_c\delta\rangle}\mathcal{S}+{}_{\langle\beta_c\delta\rangle}\theta_2}\ \Gamma_2 : {}_{\langle\beta_c\rangle}\underline{v}
\end{array}\right. \\
\Downarrow_{{}_{\langle\beta_l\rangle}\kappa+{}_{\langle\beta_c\rangle}\mathcal{E}+{}_{\langle\beta_c\rangle}\theta_1+{}_{\langle\beta_c\delta\rangle}\mathcal{S}+{}_{\langle\beta_c\delta\rangle}\theta_2}\ \Gamma_2 : {}_{\langle\beta_c\rangle}\underline{v}
\end{array}$$

For brevity we are using the symbol κ to refer to cost originating from the `let` expression, so ${}_{\langle\alpha\rangle}\kappa = {}_{\langle\alpha\rangle}\mathcal{S} + {}_{\langle\alpha\rangle}\mathcal{O}(y = \underline{e}_2)$. We know that for the transformation to be valid in the first place, $\underline{e}_1$ cannot mention y, therefore we especially have $\underline{e}_1 = \underline{e}_1[_{\langle\beta_l\gamma\rangle}\hat{y}/_{\langle\gamma\rangle}y]$ and can skip renaming.

If we do the same for the *unoptimised* expression, we obtain a slightly different evaluation tree:

$$\begin{array}{l}
\Gamma' : {}_{\langle\alpha_c\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to {}_{\langle\alpha_l\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \underline{e}_3, ...\} \\
\left[\ \Gamma' : \underline{e}_1\ \Downarrow_{\theta'_1}\ \Gamma'_1 : {}_{\langle\delta\rangle}C\ \underline{\hat{x}}\right. \\
\left[\begin{array}{l}
\Gamma'_1 : {}_{\langle\alpha_l\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \underline{e}_3[_{\langle\gamma\rangle}\underline{\hat{x}}/_{\langle\gamma\rangle}x] \\
\left[\ \Gamma'_1, \hat{y} \mapsto \underline{e}_2[_{\langle\alpha_l\gamma\rangle}\hat{y}/_{\langle\gamma\rangle}y] : \underline{e}_3[_{\langle\alpha_l\gamma\rangle}\hat{y}/_{\langle\gamma\rangle}y][_{\langle\gamma\rangle}\hat{x}/_{\langle\gamma\rangle}x]\ \Downarrow_{\theta'_2}\ \Gamma'_2 : \underline{v}'\right. \\
\Downarrow_{\langle\alpha_l\rangle\kappa+\theta'_2}\ \Gamma'_2 : \underline{v}'
\end{array}\right. \\
\Downarrow_{\langle\alpha_c\rangle\mathcal{E}+\langle\alpha_c\rangle\theta'_1+\langle\alpha_c\delta\rangle\mathcal{S}+\langle\alpha_c\delta\alpha_l\rangle\kappa+\langle\alpha_c\delta\rangle\theta'_2}\ \Gamma'_2 : {}_{\langle\alpha_c\rangle}\underline{v}'
\end{array}$$

We assume that we choose the same fresh variable name $\hat{y}$ in both worlds without loss of generality. We also assume $\Gamma \equiv \Gamma'$, from which follows that $\underline{\hat{x}}$ is indeed identical in both cases, as well as that we arrive at equivalent heaps $\Gamma_2 \equiv \Gamma'_2$. However, we have to take a closer look at costs. For reference, here are again the two profile terms we get after evaluating the expressions:

$$\begin{aligned}
\theta &= {}_{\langle\beta_c\rangle}\mathcal{E} + {}_{\langle\beta_c\delta\rangle}\mathcal{S} + \quad {}_{\langle\beta_l\rangle}\kappa \quad + {}_{\langle\beta_c\rangle}\theta_1 + {}_{\langle\beta_c\delta\rangle}\theta_2 \\
\theta' &= {}_{\langle\alpha_c\rangle}\mathcal{E} + {}_{\langle\alpha_c\delta\rangle}\mathcal{S} + {}_{\langle\alpha_c\delta\alpha_l\rangle}\kappa + {}_{\langle\alpha_c\rangle}\theta'_1 + {}_{\langle\alpha_c\delta\rangle}\theta'_2
\end{aligned}$$

As our aim is to find a way to match these profiles as closely as possible, we clearly want to set $\beta_c = \alpha_c$, as this allows us to match up all cost apart from the κ term. However, it is less clear how we should set β_l. After all, we cannot possibly set $\beta_l = \alpha_c\delta\alpha_l$, as δ is the value annotation that determines the taken branch, and therefore depends on the concrete evaluation of the scrutinee $\underline{e}_1$. This means that the concrete cause will only get determined once we run the program – which is incompatible with our intention of setting annotations beforehand!

4.8.5 Overhead

There is a good reason why we could not match up the annotations in the last section. After all, before the code transformation took place, evaluation of the `let` expression was conditional on the `case` expression taking the right branch. This is exactly what the δ annotation describes: the reason we encountered the C constructor, and therefore decided to take the given branch, leading the program to emit the `let` cost κ. Also note that we obtained the two profiles under the expressed assumption that we actually take the $\underline{e}_3$ branch. If we assume for a moment that evaluation would take a different path, the difference would be even more pronounced, as the cost κ would now completely disappear from the world W'.

What we conclude from this is that even though they are equal, we should not actually attempt to match κ from the original world with κ from the alternate world. Instead, we see them as distinct instances: The optimisation removed what caused the κ cost from the original program, and then reintroduced something else to compensate for its removal, which just so happens to also emit κ.

What do we learn from this for how to handle the code transformation? The removal is something that we do not need to track, so we can simply drop the cost from the profile silently. However, then the costs κ "re-appearing" in θ is a W event that does not exist in the alternate world W'. We conclude that these must be effects of our miracle, which is the application (or counter-factually the reversal) of the optimisation. This means that the cost becomes an optimisation *overhead* in our eyes.

What this means is that the cost κ is caused by whatever caused the compiler to float the `let` expression in the first place. Let us suppose that this was the match on the exact expression fragment ${}_{\langle\alpha_c\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \rightarrow {}_{\langle\alpha_l\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \underline{e}_3, ...\}$ [13]. This match was in turn caused by $\beta_l\beta_c$ by the logic we introduced in Section 4.3.4. From this follows that we should actually set $\beta_l = \alpha_c\alpha_l$, which gives us the completed `let` floating rule:

$$\begin{aligned} &{}_{\langle\alpha_c\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \rightarrow {}_{\langle\alpha_l\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \underline{e}_3, ...\} \\ &\qquad\Longrightarrow\quad {}_{\langle\alpha_c\alpha_l\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \big({}_{\langle\alpha_c\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \rightarrow \underline{e}_3, ...\}\big) \end{aligned}$$

What can we claim about the properties of this rule? This time, the annotation changes are substantially harder to predict. For starters, note that $\beta_l = \alpha_c\alpha_l$ gets inserted into the expression terms when substituting y by the fresh $\hat{y}$, which means that we will have extra α_c annotations in $\underline{\hat{e}}_3$ that were not present in $\underline{\hat{e}}'_3$, from which we would have to conclude that have $\underline{\hat{e}}_3 \not\equiv \underline{\hat{e}}'_3$. To prevent this, we will define our equivalence relation to not only allow replacing the optimised expression, but also adding extra α_c annotations:

$$\underline{e} \quad\equiv\quad {}_{\langle\alpha_c\rangle}\underline{e}' \qquad \text{where } \underline{e} \equiv \underline{e}'$$

This re-establishes $\underline{\hat{e}}_3 \equiv \underline{\hat{e}}'_3$, and we can show by induction over the nested evaluation that $\Gamma_1 \equiv \Gamma'_1$, $\Gamma_2 \equiv \Gamma'_2$ and $v \equiv v'$ hold as well.

Note however that this change will never actually "reach" the top-level judgement: We can argue just like we did back in Section 4.7.1 on page 95 that α_c actually "encapsulates" the effects of $\hat{y}$ already. To be concrete, $\hat{y}$ was freshly generated by our `let`

[13]Which is not strictly speaking true, as we mentioned that this transformation only applies given certain program properties. For example floating the binding up to become a global constant would be caused by $\underline{e}_2$ having no live variables. However, we have no way to encode such "negative" properties, therefore we can only ignore them.

expression, so the only way that it can influence the rest of the program would be either through the result value v or the profile θ_2. As we have annotated both with α_c, this means that all effects of the substituted variable must end up carrying this particular annotation anyway.

Apart from extra annotations, we also have substantial profile changes. The reason is not only that we might have extra instances of κ in θ, but we could also have differences due to us changing the $\alpha_c\delta\alpha_l$ annotation on the cost to just $\alpha_c\alpha_l$. Therefore we need a new profile comparison function $\leq_l$ as well:

$$\begin{array}{rcll}
\emptyset & \leq_l & \emptyset & \\
\underline{\theta} & \leq_l & \theta' + \underline{\mathcal{O}} & \text{where } \theta \leq_l \theta' \\
\underline{\theta} + \underline{\mathcal{O}} & \leq_l & \theta' + \underline{\mathcal{O}} & \text{where } \theta \leq_l \theta' \\
\underline{\theta} + \underline{\mathcal{O}} & \leq_l & \theta' + {}_{\langle\alpha_c\rangle}\underline{\mathcal{O}} & \text{where } \theta \leq_l \theta' \\
\theta + {}_{\langle\alpha\rangle}\kappa & \leq_l & \theta' & \text{where } \theta \leq_l \theta' \\
\theta + {}_{\langle\alpha\rangle}\kappa & \leq_l & \theta' + {}_{\langle\alpha\delta\rangle}\kappa & \text{where } \theta \leq_l \theta', \delta \text{ arbitrary}
\end{array}$$

This lists all the profile changes we have to allow for this particular optimisation to work: On one hand we must allow α_c annotations to appear in θ' at least for intermediate states. But on the other hand we have to also account for the profile-specific changes due to the relocation of the costs associated with the `let` expression. This means that we not only must allow κ terms to vanish, but also to gain additional annotations in W'. Note that, again, the concrete δ is only determined at runtime, so for the purpose of equivalence we must allow arbitrary changes.

This set-up is what we need in order to be able to claim that $\theta \leq_l \theta'$. And while it might appear like we had to concede a lot of rigour with our modification of the profile comparison function $\leq$, the result is actually as good as we could hope for: The annotation changes for κ just truthfully represent the decision of the compiler to change control flow. And the over-annotation of α_c is only a temporary effect that will not show up in the finished program profile. This means that yet again we have a basically perfect result: For all costs that remain after the transformation, our annotations are as accurate as they could possibly be.

At this point we should also take a moment to acknowledge that we could only obtain this result because of our choice of alternate world in Section 4.6 on page 85. After all, for a simple "crash" alternate world the `let` annotation β_l would have been seen as a cause for evaluating the contained `case` expression. This means that for example the cost $\mathcal{E}$ would get assigned with an annotation of $\beta_c\beta_l = \alpha_c\alpha_l$, whereas before it only got annotated with α_c. This would have been a much larger change, as now `let` floating would *increase* the annotations on all expressions that we "pass"!

4.8.6 Floating Case

We have seen that for the purpose of profiles, floating `let` bindings outwards work out quite nicely. However, what about if we change things around a bit by floating a one-branch `case` expression inwards?

$$\begin{array}{l}
{}_{\langle\alpha_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to {}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_2\ \texttt{of}\ \{D\ y \to \underline{e}_3; ...\}\} \\
\qquad\qquad\qquad\qquad \Longrightarrow \quad {}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_2\ \texttt{of}\ \{D\ y \to {}_{\langle\alpha_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to \underline{e}_3\}; ...\}
\end{array}$$

Here we have two `case` expressions, with one branch and multiple branches respectively. What we did is to "float" the one-branch `case` expression scrutinising $\underline{e}_1$ into the $\underline{e}_3$ branch. This is generally a very good idea where we can show that no other branch of the inner `case` expression was using x. This transformation allows these branches to skip the scrutinisation altogether.

Note that we skipped ahead a bit and already attached the "right" annotations to the above code transformations. Let us play through the evaluations anyway so we can have a look at the exact consequences this has on the profile. Here is the evaluation of the original expression, which is now our alternate world W':

$$\begin{array}{l}
\Gamma'_1 : {}_{\langle\alpha_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to {}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_2\ \texttt{of}\ \{D\ y \to \underline{e}_3; ...\}\} \\
\left[\ \Gamma'_1 : \underline{e}_1\ \Downarrow_{\theta'_1}\ \Gamma'_2 : {}_{\langle\delta_1\rangle}C\ \underline{\hat{x}} \right. \\
\left[\begin{array}{l}
\Gamma'_2 : {}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_2\ \texttt{of}\ \{D\ y \to \underline{e}_3[{}_{\langle\alpha_1\delta_1\gamma\rangle}\underline{\hat{x}}/{}_{\langle\gamma\rangle}x]; ...\} \\
\left[\begin{array}{l}
\Gamma'_2 : \underline{e}_2\ \Downarrow_{\theta'_2}\ \Gamma'_3 : {}_{\langle\delta_2\rangle}D\ \underline{\hat{y}} \\
\Gamma'_3 : \underline{e}_3[{}_{\langle\alpha_1\delta_1\gamma\rangle}\underline{\hat{x}}/{}_{\langle\gamma\rangle}x][{}_{\langle\gamma\rangle}\underline{\hat{y}}/{}_{\langle\gamma\rangle}y]\ \Downarrow_{\theta'_3}\ \Gamma'_4 : \underline{v}
\end{array}\right. \\
\Downarrow_{\langle\alpha_2\rangle\mathcal{E}+\langle\alpha_2\rangle\theta'_2+\langle\alpha_2\delta_2\rangle\mathcal{S}+\langle\alpha_2\delta_2\rangle\theta'_3}\ \Gamma'_4 : {}_{\langle\alpha_2\delta_2\rangle}\underline{v}
\end{array}\right. \\
\Downarrow_{\langle\alpha_1\rangle\mathcal{E}+\langle\alpha_1\rangle\theta'_1+\langle\alpha_1\rangle\mathcal{S}+\langle\alpha_2\rangle\mathcal{E}+\langle\alpha_2\rangle\theta'_2+\langle\alpha_2\delta_2\rangle\mathcal{S}+\langle\alpha_2\delta_2\rangle\theta'_3}\ \Gamma'_4 : {}_{\langle\alpha_2\delta_2\rangle}\underline{v}
\end{array}$$

From the correctness of the transformation we know both that $\underline{e}_2$ cannot mention x, and that $\underline{e}_1$ cannot contain a reference to y.

Let us then continue with the optimised expression:

$$\begin{array}{l}
\Gamma_1 : {}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_2\ \texttt{of}\ \{D\ y \to {}_{\langle\alpha_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to \underline{e}_3\}; ...\} \\
\left[\ \Gamma_1 : \underline{e}_2\ \Downarrow_{\theta_2}\ \Gamma_2 : {}_{\langle\delta_2\rangle}D\ \underline{\hat{y}} \right. \\
\left[\begin{array}{l}
\Gamma_2 : {}_{\langle\alpha_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to \underline{e}_3[{}_{\langle\gamma\rangle}\underline{\hat{y}}/{}_{\langle\gamma\rangle}y]\} \\
\left[\begin{array}{l}
\Gamma_2 : \underline{e}_1\ \Downarrow_{\theta_1}\ \Gamma_3 : {}_{\langle\delta_1\rangle}C\ \underline{\hat{x}} \\
\Gamma_3 : \underline{e}_3[{}_{\langle\alpha_1\delta_1\gamma\rangle}\underline{\hat{x}}/{}_{\langle\gamma\rangle}x][{}_{\langle\gamma\rangle}\underline{\hat{y}}/{}_{\langle\gamma\rangle}y]\ \Downarrow_{\theta_3}\ \Gamma_4 : \underline{v}
\end{array}\right. \\
\Downarrow_{\langle\alpha_1\rangle\mathcal{E}+\langle\alpha_1\rangle\theta_1+\langle\alpha_1\rangle\mathcal{S}+\theta_3}\ \Gamma_4 : \underline{v}
\end{array}\right. \\
\Downarrow_{\langle\alpha_2\rangle\mathcal{E}+\langle\alpha_2\rangle\theta_2+\langle\alpha_2\delta_2\rangle\mathcal{S}+\langle\alpha_1\alpha_2\delta_2\rangle\mathcal{E}+\langle\alpha_1\alpha_2\delta_2\rangle\theta_1+\langle\alpha_1\alpha_2\delta_2\rangle\mathcal{S}+\langle\alpha_2\delta_2\rangle\theta_3}\ \Gamma_4 : {}_{\langle\alpha_2\delta_2\rangle}\underline{v}
\end{array}$$

In the end we arrive at the following profiles:

$$\begin{aligned}
\theta &= {}_{\langle\alpha_1\alpha_2\delta_2\rangle}\mathcal{E} + {}_{\langle\alpha_1\alpha_2\delta_2\rangle}\mathcal{S} + {}_{\langle\alpha_1\alpha_2\delta_2\rangle}\theta_1 + {}_{\langle\alpha_2\rangle}\mathcal{E} + {}_{\langle\alpha_2\delta_2\rangle}\mathcal{S} + {}_{\langle\alpha_2\rangle}\theta_2 + {}_{\langle\alpha_2\delta_2\rangle}\theta_3 \\
\theta' &= {}_{\langle\alpha_1\rangle}\mathcal{E} + {}_{\langle\alpha_1\rangle}\mathcal{S} + {}_{\langle\alpha_1\rangle}\theta'_1 + {}_{\langle\alpha_2\rangle}\mathcal{E} + {}_{\langle\alpha_2\delta_2\rangle}\mathcal{S} + {}_{\langle\alpha_2\rangle}\theta'_2 + {}_{\langle\alpha_2\delta_2\rangle}\theta'_3
\end{aligned}$$

And again the changes do not seem too severe – the four terms originating from the evaluation of $\underline{e}_2$ and $\underline{e}_3$ have not been touched, therefore most of the profile seems to be left intact. On the other hand, that we gain extra annotations on the cost originating from the relocated `case` expression is neither surprising nor concerning: The whole goal was to change control flow in a way that evaluation of $\underline{e}_1$ now depends on the branch we take, and therefore on $\alpha_2\delta$. And as the final annotations work out to include $\alpha_1\alpha_2$, we even have a consistent causal story here if we just see the cost as "re-appearing" as we did in the last section.

4.8.7 Preemption

However, there is a more subtle problem with our argument. Specifically, suppose we try to again show equivalence between the result values, costs and heaps. Quickly we run into an issue: For the first version we evaluated $\underline{e}_1$ followed by $\underline{e}_2$ and then $\underline{e}_3$, whereas after the optimisation the order has changed to $\underline{e}_2$, then $\underline{e}_1$ and finally again $\underline{e}_3$. Now we can easily assert that evaluation of $\underline{e}_1$ and $\underline{e}_2$ cannot depend on each other on account of not using any of the x or y variables respectively. However, their evaluations can still "interact" with each other via thunk evaluations.

To see why, simply suppose that we have a variable z referring to an unevaluated cell on both the heap Γ_1 and Γ'_1, and just for the sake of the argument that we have $\underline{e}_1 = {}_{\langle\alpha\rangle}z$ and $\underline{e}_2 = {}_{\langle\beta\rangle}z$. Let us suppose that evaluating z in either world would result in costs θ_z to be emitted. This means that we would match the (Var1) rule on whatever expression we evaluate first, but (Var2) on the second. We would obtain the following profiles:

$$\begin{aligned}
\theta'_1 &= {}_{\langle\alpha\rangle}\mathcal{V} + \cdots + {}_{\langle\alpha\rangle}\theta_z & \qquad \theta_1 &= {}_{\langle\alpha\rangle}\mathcal{V} + {}_{\langle\alpha\rangle}\mathcal{S} \\
\theta'_2 &= {}_{\langle\alpha\rangle}\mathcal{V} + {}_{\langle\beta\rangle}\mathcal{S} & \qquad \theta_2 &= {}_{\langle\beta\rangle}\mathcal{V} + \cdots + {}_{\langle\beta\rangle}\theta_z
\end{aligned}$$

Clearly it has become quite hard to regard θ_1 and θ'_1 as well as θ_2 and θ_2 respectively as equivalent. In fact, due to θ_z they might arguably even be arbitrarily dissimilar! This is worrying, as in the last section we argued that putting similar annotations on these "equivalent" profile portions might be a sign that profile quality was retained. Is there a way we can fix our reasoning?

Let us take another look at the whole process from a causal point of view. Once more we are going to do a thought experiment, this time with the question: Would

θ_z get emitted if α was false? Given our example this means we need to consider alternate worlds – before and after optimisation – where we have to count on $e_1 \neq z$. Setting $e_1' = \bot$ would not gain us any useful insight in this matter, so let us instead set $e_1' = C\ x'$ for some arbitrary x'. If we now evaluate the whole transformed expression in the alternate worlds, we observe that θ_z gets emitted in either case. After all, the way we have set things up we will always still evaluate $\underline{e}_2$, which will evaluate the thunk. So by counter-factual causality analysis this tell us that α is not, in fact, a cause for θ_z!

Yet this can not be true. After all, we could just as well set $\underline{e}_2' = D\ y'$ and arrive at the conclusion that $\underline{e}_2$ had nothing to do with θ_z either. And yet there must clearly be some causal connection, after all setting both $e_1 = C\ x'$ as well as $e_2 = D\ y'$ at the same time finally makes θ_z disappear! What we have here is the well-known causal phenomenon of redundant causes preempting each other [McDermott, 1995] [14]. For the purpose of causing θ_z, we just need either α or β, meaning that if both are present, they are completely interchangeable. Hence we can actually argue that even though, for example, θ_1 and θ_1' are not equivalent, we can show that for the completed top-level profiles θ and θ' all cost either has the same cause term as before, or one where we have exchanged a cause by another equivalent one. Either is equally valid for our analysis, therefore this does not constitute a decrease in profile validity.

4.8.8 Case-Of-Case

Thanks to our careful preparations, all the examples discussed so far have been fairly well-behaved: We had fairly obvious places to put annotations, and after a bit of contemplation it was never hard to see that the resulting profiles worked out nicely in one way or the other. However, there is no guarantee of that, so we have to expect to encounter more difficulties once we go beyond simple floating transformations.

Consider an instance of the case-of-case transformation that we discussed back in Section 3.4.8 on page 40, with some annotations added speculatively:

$$
\begin{aligned}
&{}_{\langle\alpha_1\rangle}\texttt{case}\ \left({}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C \to \underline{e}_2, ...\}\right)\ \texttt{of}\ \{Dy \to \underline{e}_3, ...\} \\
&\qquad\qquad\qquad \Longrightarrow \quad {}_{\langle\alpha_1\alpha_2\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C \to {}_{\langle\rangle}\texttt{case}\ \underline{e}_2\ \texttt{of}\ \{Dy \to \underline{e}_3, ...\}, ...\}
\end{aligned}
$$

Note that we have left the annotation of the inner `case` expression empty, and instead merged all existing causes into the top-level `case` expression annotation. To see why, let us once more play through the evaluation of the untransformed expression with our

[14] Note that the literature generally states the problem the other way around – how can we identify the "true" cause in such a tricky situation? However, we essentially already implement causal chains as suggested by Lewis [2000], therefore ending up only with this relatively minor interpretation issue.

causality semantics:

$$\begin{array}{l}
\Gamma' : {}_{\langle\alpha_1\rangle}\texttt{case}\ \left({}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to \underline{e}_2, ...\}\right)\ \texttt{of}\ \{D\ y \to \underline{e}_3, ...\} \\
\left[\begin{array}{l}
\Gamma' : {}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to \underline{e}_2, ...\} \\
\left[\begin{array}{l}
\Gamma' : \underline{e}_1 \ \Downarrow_{\theta_1}\ \Gamma'_1 : {}_{\langle\delta_1\rangle}C\ \underline{\hat{x}} \\
\Gamma'_1 : \underline{e}_2[{}_{\langle\gamma\rangle}\underline{\hat{x}}/{}_{\langle\gamma\rangle}x] \ \Downarrow_{\theta_2}\ \Gamma'_2 : {}_{\langle\delta_2\rangle}D\ \underline{\hat{y}}
\end{array}\right. \\
\Downarrow_{{}_{\langle\alpha_2\rangle}\mathcal{E}+{}_{\langle\alpha_2\rangle}\theta_1+{}_{\langle\alpha_2\delta\rangle}\mathcal{S}+{}_{\langle\alpha_2\delta_1\rangle}\theta_2}\ \Gamma'_2 : {}_{\langle\alpha_2\delta_1\delta_2\rangle}D\ \underline{\hat{y}}
\end{array}\right. \\
\left[\ \Gamma'_2 : \underline{e}_3[{}_{\langle\gamma\rangle}\underline{\hat{y}}/{}_{\langle\gamma\rangle}y] \ \Downarrow_{\theta_3}\ \Gamma'_3 : \underline{v}\right. \\
\Downarrow_{{}_{\langle\alpha_1\rangle}\mathcal{E}+{}_{\langle\alpha_1\alpha_2\rangle}\mathcal{E}+{}_{\langle\alpha_1\alpha_2\rangle}\theta_1+{}_{\langle\alpha_1\alpha_2\delta_1\rangle}\mathcal{S}+{}_{\langle\alpha_1\alpha_2\delta_1\rangle}\theta_2+{}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\mathcal{S}+{}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\theta_3} \\
\qquad \Gamma'_3 : {}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\underline{v}
\end{array}$$

And for the optimised version:

$$\begin{array}{l}
\Gamma : {}_{\langle\alpha_1\alpha_2\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to {}_{\langle\rangle}\texttt{case}\ \underline{e}_2\ \texttt{of}\ \{D\ y \to \underline{e}_3, ...\}, ...\} \\
\left[\ \Gamma : \underline{e}_1 \ \Downarrow_{\theta_1}\ \Gamma_1 : {}_{\langle\delta_1\rangle}C\ \underline{\hat{x}}\right. \\
\left[\begin{array}{l}
\Gamma_1 : {}_{\langle\rangle}\texttt{case}\ \underline{e}_2[{}_{\langle\gamma\rangle}\underline{\hat{x}}/{}_{\langle\gamma\rangle}x]\ \texttt{of}\ \{D\ y \to \underline{e}_3, ...\} \\
\left[\begin{array}{l}
\Gamma_1 : \underline{e}_2[{}_{\langle\gamma\rangle}\underline{\hat{x}}/{}_{\langle\gamma\rangle}x] \ \Downarrow_{\theta_2}\ \Gamma_2 : {}_{\langle\delta_2\rangle}D\ \underline{\hat{y}} \\
\Gamma_2 : \underline{e}_3[{}_{\langle\gamma\rangle}\underline{\hat{y}}/{}_{\langle\gamma\rangle}y] \ \Downarrow_{\theta_3}\ \Gamma_3 : \underline{v}
\end{array}\right. \\
\Downarrow_{\mathcal{E}+\theta_2+{}_{\langle\delta_2\rangle}\mathcal{S}+{}_{\langle\delta_2\rangle}\theta_3}\ \Gamma_3 : {}_{\langle\alpha_2\delta_2\rangle}\underline{v}
\end{array}\right. \\
\Downarrow_{{}_{\langle\alpha_1\alpha_2\rangle}\mathcal{E}+{}_{\langle\alpha_1\alpha_2\rangle}\theta_1+{}_{\langle\alpha_1\alpha_2\delta_1\rangle}\mathcal{S}+{}_{\langle\alpha_1\alpha_2\delta_1\rangle}\mathcal{E}+{}_{\langle\alpha_1\alpha_2\delta_1\rangle}\theta_2+{}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\mathcal{S}+{}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\theta_3} \\
\qquad \Gamma_3 : {}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\underline{v}
\end{array}$$

Which means that we arrive at the following cost sum:

$$\begin{aligned}
\theta &= {}_{\langle\alpha_1\alpha_2\rangle}\mathcal{E} + {}_{\langle\alpha_1\alpha_2\delta_1\rangle}\mathcal{S} + {}_{\langle\alpha_1\alpha_2\delta_1\rangle}\mathcal{E} + {}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\mathcal{S} + {}_{\langle\alpha_1\alpha_2\rangle}\theta_1 + {}_{\langle\alpha_1\alpha_2\delta_1\rangle}\theta_2 + {}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\theta_3 \\
\theta' &= {}_{\langle\alpha_1\alpha_2\rangle}\mathcal{E} + {}_{\langle\alpha_1\alpha_2\delta_1\rangle}\mathcal{S} + \quad {}_{\langle\alpha_1\rangle}\mathcal{E} \quad + {}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\mathcal{S} + {}_{\langle\alpha_1\alpha_2\rangle}\theta_1 + {}_{\langle\alpha_1\alpha_2\delta_1\rangle}\theta_2 + {}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\theta_3
\end{aligned}$$

Which once more is a very close match, especially if we consider that we have no change of annotations for θ_1, θ_2 and θ_3, which are likely to contribute the lion share of cost in a real program.

On the other hand, note that the cost of the `case` expression scrutinising the evaluation result of $\underline{e}_2$ has gained an extra $\alpha_2\delta_1$ annotation. As before, this is a direct result of the transformation reorganising the control flow in order to push the `case` expression in question inside the branches. Unlike last time however, this new annotation does not actually tell us anything useful: The program would execute a copy of the `case` expression no matter which path we take. Hence $\alpha_2\delta_1$ is for all intents and purposes not actually a cause for the cost in question. Consequently, transforming the annotated program as proposed will actually cause a slight drop in profile accuracy!

This might not seem like a big deal at this point, but note that most code transformations are applied many times during the optimisation phases, therefore we should make sure that we do not accept this without good reason. Unfortunately, our options are quite limited: Clearly we would not want to remove annotations on θ_1, θ_2 or θ_3, as that would have even more severe consequences.

The most interesting of the bad options would be to get “tricky”:

$$\begin{aligned} {}_{\langle\alpha_1\rangle}\texttt{case}\ \left({}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C \to \underline{e}_2, ...\}\right)\ \texttt{of}\ \{Dy \to \underline{e}_3, ...\} & \\ \Longrightarrow \quad {}_{\langle\alpha_1\rangle}\texttt{case}\ {}_{\langle\!\langle\alpha_2\rangle\!\rangle}\underline{e}_1\ \texttt{of}\ \{C \to {}_{\langle\alpha_2\rangle}\texttt{case}\ \underline{e}_2\ \texttt{of}\ \{Dy \to \underline{e}_3, ...\}, ...\} & \end{aligned}$$

Which would yield us the following profile:

$$\theta = {}_{\langle\alpha_1\rangle}\mathcal{E} + {}_{\langle\alpha_1\alpha_2\delta_1\rangle}\mathcal{S} + {}_{\langle\alpha_1\alpha_2\delta_1\rangle}\mathcal{E} + {}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\mathcal{S} + {}_{\langle\alpha_1\alpha_2\rangle}\theta_1 + {}_{\langle\alpha_1\alpha_2\delta_1\rangle}\theta_2 + {}_{\langle\alpha_1\alpha_2\delta_1\delta_2\rangle}\theta_3$$

This would allow us to match the ${}_{\langle\alpha_1\rangle}\mathcal{E}$ profile portion through the back door by comparing costs of the `case` expression scrutinising $\underline{e}_1$ with the one scrutinising $\underline{e}_2$. While this might look good on paper, it is a bad idea – the assumption that the two `case` costs are comparable is questionable at best. A simple extension of our performance model would invalidate this idea.

In the end, we have to admit that we simply cannot retain perfect profiles for every code transformation within the confines given by our causality model. Our only consolation at this point is that we can argue that the case-of-case transformation is typically a pretty clear one-way optimisation, so we would expect that the number of applications in an optimisation pass is expected to be relatively low.

4.8.9 Rules

We have seen that we cannot always preserve the accuracy of our profile when running our code through transformations. So how bad would our profiles get in the “worst case”? For our purposes, the most general code transformation we want to support are programmer-supplied custom optimisation rules [Peyton Jones et al., 2001].

Recall the `fold/build` rule we demonstrated in Section 3.4.1:

$${}_{\langle\alpha\rangle}\texttt{foldr}\ k\ z\ \left({}_{\langle\beta\rangle}\texttt{build}\ g\right) \quad \Longrightarrow \quad {}_{\langle\alpha\beta\rangle}g\ k\ z$$

Assessing this rule is difficult. After all, both `foldr` and `build` are non-trivial functions, the effects of which we can not easily tease apart like we did for simple expression fragments in the previous sections. So unless we are willing to look deeply into their definitions, these functions are simply black boxes to us. Apart from the fact that the evaluation depends on both α and β and that they make use of sub-expressions k, z and g there is very little we can actually know.

Therefore, we are forced to assume that the optimisation changes all costs that it could possibly reach. This means that we assume a "worst case" behaviour for the program as it was before code optimisation, namely that evaluating the expression in question caused no costs at all (or $\bot$) while still producing an equivalent return value. Our miracle would for example be the judgement:

$$\Gamma : \texttt{foldr}\ k\ z\ (\texttt{build}\ g)\ \Downarrow_{\bot}\ \Gamma' : v$$

By this point we should be quite familiar with how this kind of thing turns out: If we assume this cost term in our alternate world, then every cost produced by the main-world judgement of $g\ k\ z$ must be an effect. Just like with overheads, their cause will be the same as what caused the optimisation. As we know that rules getting applied depends exactly on the compiler spotting the desired expression pattern, we also know that the transformation was caused exactly by the causes annotated on the matched expression. In our example, this would be $\alpha\beta$.

To get the appropriate annotations on the effects we have to again employ indirect measures, with all their imperfections. By consulting Figure 4.19 we find that in our instance, the most straight-forward way of annotating all cost generated by the modified part of the program is simply to put an annotation on the application $g\ k\ z$ itself. Note however that this will not generally work due to `let` and `case` expression annotating their results differently. Fortunately, we have already solved this problem in Section 4.8.2 using push annotations, so we can simply re-use this here. So let us say that we have a rule matching an expression pattern with unknowns $\underline{e}_1, \underline{e}_2 ...$ and transforming it to a new expression $opt(\underline{e}_1, \underline{e}_2, ...)$. The compound pattern match event will depend on a number of annotations $\alpha\beta\cdots$ as explained in Section 4.3.4, which we can then re-annotate as the cause of the optimisation:

$$_{\langle\alpha\rangle}\cdots\underline{e}_1\cdots{}_{\langle\beta\rangle}\cdots\underline{e}_2\cdots \quad\Longrightarrow\quad {}_{\langle\!\langle\alpha\beta...\rangle\!\rangle}\,opt(\underline{e}_1, \underline{e}_2, ...)$$

To reiterate, this is the most conservative way we could annotate given the expression properties we know. Note that we could be over-annotating costs here, after all it would not be too hard to construct an example where, say, β becoming false would not cause a certain profile portion caused by α to change. However due to the limited amount of information we have to accept this sort of annotation overreach. To make matters worse, note that by using push annotations we might also be adding additional annotations on the return value. Given that we assume that the result value does not change, this is almost automatically a reduction in annotation usefulness.

However, things are not all bad. Optimisations often work on functions that are coordinating heavily, so it is in fact quite hard to find real-world examples for rules where the results depend completely on the whole pattern matched. In the example of the short-cut rule, we are clearly reducing the number of expressions involved, so it is not hard to see that bringing together the annotations at the only remaining expression is our only choice. And simply by looking at the costs involved, we can even determine that in our example we have not actually introduced any sort of profile degradation.

4.8.10 Final Notes

By this point we have covered the whole spectrum of optimisations present in the compiler. It should be clear that no matter the code transformation, will find some kind of annotation scheme that guarantees us to never lose annotations. In the worst case we can always fall back to just push-annotating every cause of the optimisation on top of the transformed expression, which as explained in the last section is always valid.

However, the more information we have about the transformation, the better a job we can do. This is especially true where we can anchor the logic of the transformation into our causality model as we did with `let` and `case` expressions. However, let us take a moment to review all the different types of profile effects we have identified, from benign to more severe:

1. Firstly we might have cost reductions, as shown in Section 4.8.1. These are both what optimisations want to achieve, as well as easy for us to handle, as we are only interested in costs that actually appear in the final program.
2. Another profile change that we can gloss over is preemption as investigated in Section 4.8.7. Here we simply have cost and value terms switching out redundant causes, which means that we neither gain nor lose actual diagnostic accuracy.
3. In Section 4.8.5 we introduced the other side of the coin: overheads. Where a transformation makes a conscious decision to trade one cost for another, we have to acknowledge that fact by annotating the cause.
4. Finally, in Section 4.8.8 and Section 4.8.9 we encountered the most problematic change we were forced to make: Annotations overreaching. This was a matter of us hitting the limits of what causal stories we can express using the tools given to us by the evaluation.

In the end, we expect that the causality model will be able to enumerate all possible causes for every effect, while doing a reasonably good job at not introducing too much noise on top. As we explained at the beginning of this chapter, this is all we could hope to achieve. We will show in Chapter 6 that in practice we can actually tolerate false positives to a significant degree.

Chapter 5

Profiling

"This hourglass keeps very good time. Helvetian made! But you have to watch it. Whenever I shout 'cuckoo' it is time for all the hotel guests to turn their hourglasses around."

— *Asterix in Switzerland, René Goscinny & Albert Uderzo*

At this point we have developed a sound theoretical foundation for evaluating profiling designs. To be concrete, what we have is a working model of how we can take a Haskell program, desugar it into our annotated abstract language, and then proceed to optimise and evaluate it. The result will not only be the program result, but also a perfect correct-by-construction cause-annotated abstract cost profile: A full record of the cause behind every abstract cost emitted.

While developing this model we took great care to reflect real-world compilation and execution of Haskell code as closely as possible. The reason is that we go into this chapter with the intention of mapping the abstract model back to actual Haskell execution. This means that we are going to translate every element of our model back into its real-life counter part: Abstract costs will become real costs, abstract evaluation will reflect the real program evaluation, and cause annotations will be references to the source code of some form. As we have shown that our abstract model is consistent, this means that if we find cost in our program that corresponds to our performance model we only have to ask what cause term we would assign to the abstract cost in our model. As the cause term corresponds to a full history of how the source code influenced cost generation, the result would ideally be a "perfect" explanation of how the cost in question was emitted.

However, there is a very good reason why we limited our considerations to an abstract model in the first place: For a concrete profiling solution our design space is actually heavily restricted. Instrumentation and sampling will generally only be able to tell approximately how much cost was generated from each part of the program.

This means that our mapping to abstract evaluation will be fairly approximate: We can only ever hope to reconstruct very limited parts of the "actual" cause terms. To make matters worse, we also have to start thinking about how exactly we are going to present all of our data to the user. After all, in the end this is about causal analysis, which means that we ought to break our data down using suitable verbs, nouns and explanations.

We will start this chapter with reviewing our requirements and settling on a concrete design for our profiling solutions. Section 5.1 will serve as an introduction to this, while Section 5.2 will review the kinds of performance data we wish to collect. Section 5.3 on page 121 will then round out our design with the most important part: How much of the cost annotations we can reconstruct, as well as what user intuitions these might correspond to. We will especially take this opportunity to evaluate other profiling approaches with respect to our abstract model.

Starting with Section 5.4 on page 129 we will tackle concrete implementation concerns. We will begin with explaining how to adapt GHC's passes to maintain source code annotations that reflect cause annotations from the last chapter. In Section 5.5 on page 143 we will show how we can continue to track these annotations even after GHC has translated the functional Core code into the imperative Cmm language. This only leaves actual code generation, where Section 5.6 on page 149 will explain how we can embed our debugging data into executables using a mix of standardised DWARF code and GHC-specific extensions.

By that point, we will have learned how we can generate the program complete with enough annotations to track virtually any cost back to the source code. However, in order to solve its performance problems, we still need to collect and interpret performance data. To this end, Section 5.7 on page 162 will explain how we can extend the run-time system of programs compiled with GHC to collect samples telling us about a variety of performance metrics. Finally, Section 5.8 on page 168 will describe how we can enable the user to analyse and evaluate our collected data.

5.1 Design

Let us quickly remind ourselves what the profiling task is according to our earlier definition in Section 2.1 on page 5. The whole point of our solution is to allow the user to map an abstract notion of performance (*verbs*) to an abstract notion of causes (*nouns*). As a simple one-to-one mapping seldom contains enough information to allow the user to fully understand the problem, it is furthermore a good idea to give the programmer an idea exactly how the program executed. Such *explanations* are vital for effective program analysis, and should ideally strike a balance between saying enough about the program run to analyse performance, and overwhelming the programmer with too much "incidental" information. In either case, we need statistics that give us an idea of roughly how much cost we are tracking at each point during execution, in order to allow the user to focus their search.

So what will be the nouns, verbs and explanations for our profiling solution? Let us start with the easiest part: Given our groundwork from the last chapter, our causes and therefore nouns will be referring to the program source code. We can easily generate these causes simply by remembering the original source code spans while generating Core code during parsing and desugaring. And while maintaining these source code links throughout compilation will require a few compilation changes, we can implement this with only modest overheads. On the other hand, we have a much broader set of choices when we consider how to collect performance data. We could go by time consumption, processor cycles, or attempt to pinpoint certain hardware processes such as cache misses. Taking inspiration from the performance model we developed in Section 3.5, we also have the option to specifically instrument Haskell code characteristics, such as stack or heap usage. Collecting such data will involve working alongside a program execution, which comes with its own set of challenges.

Finally, for explanations we will mainly pursue two approaches. Our main idea will be a pretty common one for profiling tools: We will be able to derive a good number of source code links using our abstract model, but this does not explain very well *how* the given source code element ended up influencing performance. Therefore we will take apart the full cause term in a way reminiscent of call stacks. Furthermore, we will show how we can use intermediate compilation results to allow the user to discover what transformation happened in the compiler. In either case, our implementation will attempt to provide this information without having to do any information gathering on its own. This means that our explanatory data is basically "free", and especially allows us to in principle work with any given sampling back-end.

5.2 Metrics

Let us first talk about how we approach performance measurements. Due to our work on the performance model in Section 3.5 on page 41 we generally have a pretty good idea about how the program generates cost. Yet as we argued in Section 2.2.4 on page 12, this is not enough for profiling: We need to be able to actually quantify these costs. To make matters worse, different verbs will require entirely different performance metrics as well as sampling approaches. For example information collection on low-level verbs will generally mean that we need to interface with the CPU and the operating system, while Haskell-specific metrics will be measured by the run-time system.

Furthermore, given our approach it is pretty clear that we are highly interested in high-quality performance data. After all, we have put considerable effort into making sure that our view of causality is consistent even after program optimisations have been applied. This means that our profiling approach is especially suitable for working with heavily optimised programs, where our main interest might be squeezing out the last few percents of performance. Therefore it is important that we proceed in a disciplined fashion and make sure that our approach is able to even pick out relatively small regressions in performance.

5.2.1 Skews

Most importantly, we need to make sure that we do not introduce systematic errors into our profiling: If our performance measurement approach ends up selectively increasing the cost associated with certain program elements, we might end up seeing performance problems where they normally do not exist. This is a very real danger, as gathering performance metrics has a cost of its own. In the extreme case, we might end up profiling our own profiling solution instead of the program.

There are two approaches we can use to prevent this. The easiest way is to simply make sure that we have as little overhead as possible. After all, where our sampling is so cheap that it has no significant influence on overall program performance, it cannot have a significant influence on profiling results. However this is slightly dangerous for us to rely on, as what we mean by "significant" can quickly change. For instance as we noted above, we want to be able to identify relatively minor influences on performance. Furthermore, sometimes we might be forced to increase the overhead of our data collection, for example because a short program run time forces us to collect a lot of samples over a relatively short time period.

In these situations we can attempt to prove that our profiling will have few skews *despite* having overhead. Fortunately, this is quite often the case: For example, taking heap residency profiles on garbage collection will introduce basically no skews despite its moderate cost. After all, garbage collections taking longer is unlikely to have a big impact on how the program executes. And even for low-level measurements we can often prevent skews by leaving most of the work to, say, the operating system, which is outside of the scope of our performance measurements.

5.2.2 Time

Let us then start considering the different verb implementations. The most basic verb for us is simply time passing, which is the resource that basically every abstract cost will indirectly consume. Especially most of the "constant" costs for applying a rule (such as $\mathcal{C}$, $\mathcal{L}$, $\mathcal{A}$, $\mathcal{V}$, $\mathcal{U}$ and $\mathcal{E}$) will be covered by no specialised verbs, therefore it is always vital that we can measure this verb in one form.

Yet this does not mean that profiling by time is without design choices. For starters, we have to choose what program phases we want to consider: For example, a standard restriction for Haskell programs is to only consider "mutator" time, which is the program run time excluding garbage collections. As we are primarily interested in the performance of our own compiled code, this makes sense most of the time. After all, it is generally better to analyse garbage collection efficiency using specialised verbs such as allocation or residency, which we will cover in Section 5.2.4 forward.

Furthermore, there might be other outside influences that might cause time loss, which we might or might not want to discount. For example, our process might get de-scheduled by the operating system because of multi-tasking, or to wait for an I/O request to complete. Whether we see these as performance problems depends on whether we see these as outside influences or expect our program to have control over them.

5.2.3 CPU

Moreover, where appropriate we can even break down mutator time further. After all, the code produced for our Haskell program will run on a CPU, which is a complex piece of hardware in its own right. This means that even beyond the relatively simple performance model we derived for Haskell programs, there are subtle performance characteristics to tease out. For example, depending on the structure of our control flow and memory access patterns, the processor might spend extra cycles waiting for its tables to reload on a mis-predicted branch. In a similar fashion, accessing an obscure memory region might cause a cache miss followed by a slow reload.

For capturing statistics about these kinds of verbs we have to generally depend on hardware support. However note that once we have located the source of the slowdown, we can use the very same mechanisms to track them back to the source code that we always employ: After all, for our purposes for example a mis-predicted jump could simply be a part of the abstract $\mathcal{C}$ cost for a `case` expression. Therefore it will be indirectly tracked by our abstract semantics, which yields us cost terms just like for any other cost discussed here.

5.2.4 Allocation

The reason that we are interested in low-level statistics such as cache misses is of course that memory access can have substantial impact on program performance. We stressed this point quite a bit when we developed how our performance model would handle memory. Our first verb of note here is allocation, which corresponds to the act of introducing new data into our memory system.

However, we have two different forms of allocation that we might be interested to track: Let us first discuss the faster options, which is stack or – for our purposes synonymously – register allocation as discussed back in Section 3.5.3 on page 46. The reason that this option is fast is that we have very predictable access patterns, which means that the top of the stack generally remains in low-level processor caches for a long time. However, significant allocation on the stack can still be a performance problem, as for example the factorial function we discussed back in Section 3.4 on page 30 had $O(n)$ stack complexity. Where a program consumes this amount of stack space capturing appropriate measurements would definitely be useful.

The second important form of allocation that a Haskell program performs is heap allocation. For allocation purposes this is actually remarkably cheap, as the initial heap allocation takes place in a nursery memory block, which is also very likely to reside in processor caches. Given efficient garbage collection this means that programming with short-lived heap objects can be remarkably fast [Sansom and Peyton Jones, 1993]. However, not all objects are short-lived enough and will eventually cause complexity by wandering into older heap generations. Furthermore, excessive heap allocation can often be a sign that optimisations failed to work properly, such as missing ways to unbox primitives as discussed in Section 3.4.2 on page 31, or failing to eliminate a closure using let-no-escape as explained in Section 3.5.11 on page 54.

5.2.5 Residency

The problem with heap objects that fail to die in the nursery is that they might ultimately lead to heap residency: Heap data that accumulates over the program run. After all, an object on the heap will be kept alive as long as there is a possibility for the program run to access it. Keeping a lot of data around in this way is generally a bad idea for performance, as not only makes memory access slower due to lost locality, but also makes major garbage collections more costly overall. Therefore we will not only see heap allocation as a verb, but heap residency as well.

However note that just like before, we can also attempt to partition the heap residency verb by the concrete cause for the residency. As Röjemo and Runciman [1996] show, we can actually see residence as being composed of four different effects – objects getting created but not accessed (*lag*), objects getting actively accessed (*use*), objects getting kept alive after the program does not need them anymore (*drag*) as well as objects never getting used (*void*). All of these correspond to different aspects of performance problems relating to heap residency: "Lag" means that we might be allocating objects too early, dominant "use" might point towards work not finishing up with the object quickly enough. Finally, "drag" and "void" correspond to the classic memory leak problem, where we keep references around for too long.

Especially note that controlling references in this fashion is quite hard in practice. After all, due to Haskell's lazy evaluation, the compiler will attempt to delay evaluation as much as possible, which – as shown in Section 3.5.8 – will cause us to capture all live variables. For the inexperienced Haskell programmer, such references can be somewhat hard to predict, and therefore often lead to tricky performance problems. As our work is however not primarily about heap profiling, we will not attempt to go too deeply into these problems, and will instead limit our approach to measuring and – ideally – pin-pointing heap allocation as well as basic heap residency.

5.3 Explanations

As the last section should have shown, there are quite a few useful performance metrics that we could use to inform program optimisation. In theory, all we need to do now is to actually implement the equivalent of our annotations in order to derive suitable cause terms. However, showing a causal connection is only half the battle in practise. After all, our cause terms are only correct in the sense that the mentioned causes can be shown to be sufficient for producing the given effect. This does not mean that the produced compound cause term is especially descriptive of the exact mechanism that led to the observed effect.

In fact, it is not hard to see the practical problem with full cause terms: They will most likely turn out to be absurdly large for any program of non-trivial size. Just consider the optimisations we investigated in the last chapter: For basically every considered rule there was a case where we had to put all visible cause annotations on one cost or the other. And this is hardly surprising, as programs generally work incrementally by building on top of the information that has been derived so far. Whether it is actual data or control flow – every action the program takes will depend on a significant chunk of the history that led up to the point in question. If we want to picture the extreme case, we could easily see all our cause terms approaching a maximal cause term covering the entire non-dead code base[1].

Such cause terms would not only be really hard to work with, but would also be prohibitively expensive to construct. Consequently our goal for this section will be to identify ways to decompose our cause terms into its components. We will start out with explaining what the theoretical basis for annotation decomposition is, followed by mapping these to intuitions borrowed from existing profiling solutions. We will then proceed explaining our actual solution options, which will generally attempt to reconstruct as much causal information as possible from a bare minimum of sampling data.

[1] Which could be useful if coverage checking was our goal [Gill and Runciman, 2007].

5.3.1 Noun Stacks

So what can we do about giant cause collections implied by our abstract model? We know that cause terms grow by getting combined from existing ones. This means that there must be a composition history behind every single cause term used in the program. Specifically, we can interpret the cause history as a tree of compositions, with each node corresponding to a cause term considered at some point during evaluation. This means that the deeper we go into the tree, the more likely it will be that the given cause term is, in fact, shared by other cause terms. After all, the extreme case here is the evaluation of the "`main`" function, from which all cost in the program *must* derive. After all, not evaluating this function means that zero costs would get emitted!

On the flip-side, this means that a primitive cause is more likely to be a significant part of our cause term if only few compositions happened since it was introduced. So for example let us say that we saw a certain `case` expression causing significant performance problems. Then we would probably check first whether the `case` expression itself was carrying an insightful annotation. Failing that, we would continue asking for the cause behind the expression getting evaluated: Where did we call that function from? What caused us to take that branch? This corresponds directly to how our cause terms was composed in the first place.

A popular approach is to focus entirely on the "control flow", which by our logic is a portion of the full cause terms. Just consider our semantics from Figure 4.19 on page 94 – we could easily reduce it to considering only control flow by disregarding all α annotations coming from values. Especially note that if we ignore for a moment that `let` and `case` expressions as well as optimisations can change expression annotations, every expression annotation would directly conform to a source code location. Therefore our cause term would directly encode a path of expressions leading back all the way to the program entry point. If we map these expressions back to the containing functions, we would essentially get a typical call stack as known from profiling imperative programs (see for example Graham et al. [1982]).

5.3.2 Lexical Scopes

This is not a new insight, and is commonly exploited by established profiling solutions. For example, we can view cost-centre profiling as making relatively coarse cause statement about lexically contained expressions. Then the approach taken in Sansom and Peyton Jones [1995] is essentially to regard the "closest" cost-centre we can find along the control flow as the one being responsible for the cost. The PhD thesis of Sansom [1994] further extends this to cost centre stacks, which attempts to capture *all* cost-centres found on this path.

$$\begin{array}{l}
{}_{\langle\alpha\rangle}\texttt{let}\ \{f = \dots \\
\qquad h = {}_{\langle\gamma\rangle}\texttt{case}\ y\ \texttt{of}\ \{ \\
\qquad\qquad C\ x \rightarrow f; \\
\qquad\qquad D\ y \rightarrow \texttt{let}\ \{\ g = \dots\}\ \texttt{in}\ g\} \\
\quad \texttt{in}\ {}_{\langle\beta\rangle}h\ x
\end{array}$$

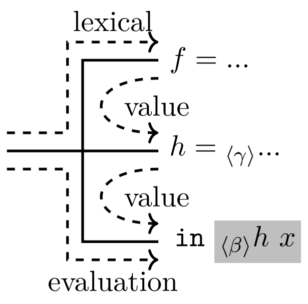

lexical
$f = \dots$
value
$h = {}_{\langle\gamma\rangle}\cdots$
value
$\texttt{in}\ {}_{\langle\beta\rangle}h\ x$
evaluation

Figure 5.1: Scopes for an application expression

However in practice it is quite hard to say what the most useful interpretation of "control flow" is. For thunks and function applications Sansom [1994] actually advocates "lexical" or "static" scoping, where we only consider the control flow path to where the closure was *allocated* instead of where it got *called*. Consider for example Figure 5.1: Here the lexical scope of f is α, and the lexical scope of g similarly $\alpha\gamma$. However, the control flow – or "evaluation" scope – will always go through the call site $h\ x$, and therefore be annotated with γ. The "full" cause term for the cost would of course be $\alpha\beta\gamma$. After all, for the (App) rule in our semantics from Figure 4.19 on page 94 this cost would be θ_b, and it gets annotated with both the cause annotation of the closure (including $\alpha\delta$) as well as β for the application expression itself.

Note that our cost causes are therefore the union of both the evaluation scope as well as the lexical scope. This is in fact always the case. To see why, note that the paths of the lexical scope and the evaluation scope always join sooner or later in order to transmit the closure reference. In Figure 5.1, this is the top-level `let` expression, and as the sketch shows, both evaluation and lexical run in parallel beyond this point. This means that the path from the start of program evaluation to the "join point" must be a part of the cost annotation. The key observation here is that the rest of the path corresponds to how the closure *value* travelled back to the join point. Fortunately, we know that outside of `let` and one-branch `case` expression every rule match on the way will add its expression annotation α to the returned value.

Furthermore, we know that these annotations will appear on the cost generated from the evaluation of the function application. This can be seen either by relying on the correctness of our causality model, as clearly not returning a reference to the closure would cause the program to not evaluate the closure. Or we could argue along the lines of Section 4.7.1 that our annotated rules respect annotation encapsulation, asserting that if we put an annotation on the only value containing a reference to our closure, evaluation must eventually go back through it, causing the cause in question to be annotated on costs.

5.3.3 Evaluation Scopes

Lexical scoping is often a good idea when making sense of a program written in a lazily evaluated language. The reason is that whether or not a given thunk gets evaluated is a rather binary choice, and more often than not the answer is "yes, eventually" [2]. The actual evaluation site therefore becomes of little interest to the profiling process.

On the other hand, the situation is a bit different for function applications: functions can get called any number of times, therefore tracking the cause behind the function application is typically a lot more interesting. Ultimately there is a reason for its popularity in conventional profiling tools. However, it is quite hard to conclusively argue one way or the other. After all, Haskell code often uses higher-order programming in order to structure its control flow, for example when using monads [Wadler, 1992]:

```
getName = {-# SCC "getName" #-}
          do putStrLn "Who␣might␣you␣be?␣"
             name ← getLine
             putStrLn $ "How␣interesting,␣" ++ name ++ "!"
```

Listing 5.1: Monad Usage Example

Here we are using the IO monad in do notation [Peyton Jones and Wadler, 1993] in order to hold a bare-bones conversation with whomever might appear on the other side of the standard terminal. Note we wrote the above program by spelling out a group of effectful statements that will execute in sequence. This monadic style has proved quite useful in preventing general awkwardness when writing Haskell programs [Peyton Jones, 2001]. In our case, it abstracts away the need to pas around a "state" token that actually ensures the sequence of I/O actions. For example unfolding and eliminating newtype wrappers would yield:

```
getName = {-# SCC "getName""␣#-}␣λs0␣→
␣␣case␣putStrLn#␣"Who might you be? "␣s0␣of␣(()␣␣,␣s1)␣→
␣␣case␣getLine#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s1␣of␣(name,␣s2)␣→
␣␣putStrLn#␣("How interesting, "␣++␣name␣++␣"!")␣s2
```

Listing 5.2: Monad Usage Example – unfolded

As useful as this style is for writing programs in an imperative style, this makes it tricky to reason about the code in terms of control flow. After all, the getName cost centre now only covers the allocation of the lambda, but the actual call to the "body code" will come from outside once we have acquired an appropriate state tag. This means that according to evaluation scoping, the execution stack for this code would actually not mention the getName cost centre at all!

[2] At which point it probably should not be a thunk in the first place, but I digress.

The trouble here is that these functions behave quite differently than we speculated earlier: Generally, when we "call" an IO function such as getName, we expect it to get passed exactly one state token. Just like with thunks, this means that we are steering the control flow not by calling functions, but by determining *what* functions to call. Such higher-order programming practises are therefore inherently incompatible with pure evaluation scoping.

To help with this, Marlow [2012] suggested to combine the best of both worlds by merging the stacks given by evaluation and lexical scoping. This is in fact how cost-centre scoping is currently implemented in GHC. The idea should be fairly familiar to us at this point: As both evaluation and lexical scope are sub-sets of the full cause term, using a super-set of them is more likely to recover relevant cause terms. In fact, we can liken this approach to how we put both α and $\overline{\beta_i}$ annotations on the cost emitted by the called code in the (App) rule from Figure 4.19 on page 94.

However note that the combination of evaluation and lexical scoping still does not yield a complete cause term by our standards. After all, if we again consider the example from Figure 5.1 on page 123 we observe that for calling f neither lexical nor evaluation scope would actually mention γ. The reason is that in our model β_0 will get updated along the way to record the history of how we determined the exact closure value to call. And while some of this will likely be part of the lexical scope, the example should demonstrate that this is not always the case. Therefore we have to note that strictly speaking, even this improved version fails to tell the full causal story.

5.3.4 Static Context

The last sections have discussed how we can break our cost annotations down into more accessible explanations for the user. However, on the way we made a number of simplifying assumptions, such as absence of optimisations and perfect information about how we arrived at a cause term. This made it a rather academic discussion. For a real-world implementation, actually obtaining information about cause terms will be tricky enough, to say nothing about their composition history. Cost-centre profiling as discussed in the last section actually needs to constrain optimisations and do non-trivial run-time instrumentation just to obtain consistent lexical and evaluation stacks. For our purposes, we would like to avoid doing either, so how close can we get under these circumstances?

Fortunately, just knowing where program evaluation was at a given time already allows us to say a lot about cause terms. Consider for instance:

$$_{\langle\alpha\rangle}\texttt{let}\ \left\{ f = {}_{\langle\beta\rangle}\lambda x.\underline{e}_1 \right\}\ \texttt{in}\ \underline{e}_2$$

Given that we find a program in the process of evaluating e.g. $\underline{e}_1$, we can make a number of assertions about what annotations our cost is going to receive, and how. Just from the fact that our expression is contained inside a lambda expression, we know that there must have been a function application using a closure derived from f. As this closure will still carry the β annotation, it follows that all cost produced by the function would get annotated with at least β directly following the return. We furthermore know that cost would gain an α annotation on the same occasion, as the (Let) rule would propagate α to usage sites of f, which the program will have to evaluate in order to obtain a reference to the closure in the first place. What we are essentially saying is that once we know where evaluation was at a certain point, we can always reconstruct at least the lexical scope from that piece of information. And as our explanation of lexical scopes in Section 5.3.2 should have shown, this is a useful subset of our full cost annotation.

It is possible to infer even richer explanations if we are willing to consult with the user at this point: The user might not only be able to judge how relevant various pieces of the lexical scope are, but also make educated guesses about past control flow using knowledge about the source program. Therefore giving the user an idea of the structure of the compiled program at the point of interest can actually be very beneficial. Simply providing a view of Core enhanced with source code annotations can allow the user to form a much more well-rounded opinion about the situation at hand. However, placing this much interpretation burden on the user is obviously not something that we should do by default, so we will only implement this as a technique for those situations where we need to drill deep into a performance problem.

5.3.5 Quality Considerations

Our approach to determining cause terms might seem to be minimal to the extreme: After all, where cost-centre profiling collects entire execution and/or lexical scopes, we have just limited ourselves to only what we can reconstruct from the lexical scope of the transformed code and user intuition! We have to confess that this is a lot less powerful than what we can do with cost-centre profiling: Using just this approach we will never be able to tell where function calls were coming from. In the worst case, we might imagine getting stuck with a completely useless nugget of information, such as that some low-level function consumed disproportional amounts of resources.

However, we are basically going to accept this risk, because it allows us to reap significant benefits in other areas. Let us compare against cost-centre profiling again: The stated goal of Sansom and Peyton Jones [1995] is to produce predictable cost-centre stacks no matter whether the program was optimised or not. While this property makes the results more expressive, the restrictions it needs to put on optimisations are actually fairly severe. After all, it is generally impossible to retain perfect stack consistency throughout arbitrary code transformations. Take for example floating inwards:

```
let f = e1 in {-# SCC ... #-} ...
  ⟹
scc<...> let f = e1 in ...
```

Listing 5.3: Cost-Centre Float-In

consider the cost for evaluating `e1`: This transformation means a clear change to the lexical scope, which we could not undo, therefore the only choice is to forbid this transformations for cost-centre profiling. For our profiling framework, we would like to retain such optimisations in order to make sure that our measurements correspond closely to real-world performance.

What is more, cost-centre profiling has to work with more instrumentation cost for managing stacks than we would be willing to accept. Each time a new cost-centre gets entered, we need to either find or allocate some sort of structure corresponding to the new noun combination. This is not trivial to implement, especially when we consider that we might have to spend extra effort to "compress" deep stacks in order to be able to reason about deep recursion. To make matters worse, maintaining lexical scoping means that we need to determine the scope of a thunk when it enters evaluation. As there is no way to know this statically, the profiling implementation consequently needs to store a cost-centre-stack representation with every single allocated closure [Sansom, 1994]. Such a change in heap layout has to be propagated into all other components working with closure objects – which means that we not only need a version of the run-time system specialised for usage with profiled code, but also one for all linked Haskell libraries.

When all is said and done, our approach of deriving information purely from statically known properties is substantially cheaper and consequently better suited for low-overhead profiling. We also have fairly good reason to suspect that the information derived using our approach will not be completely useless: For example we know that the derived cause term sub-set will always be quite relevant to a local performance problem. This directly follows by the logic introduced in Section 5.3.1: We will have perfect information about the "newest" annotation added, as well as some clues about recent annotation sources in the form of parent expressions. Furthermore, paradoxically allowing more optimisations to run might actually be beneficial for our purposes. After all, every time we inline a function to its call site, this means that the calling context will become statically associated with the called code. This means that while the worst case is still quite possible, we have reason to believe that in reality we will often have a lot more contextual information to work with.

5.3.6 Stack Tracing

If we are unsatisfied with the amount of information we can extract from the program and are willing to pay some extra cost, we always have the option of looking a bit beyond just the current evaluation position. After all, Haskell execution maintains a stack, which not only contains our working data and parameters, but more interestingly code pointers to places that we are planning to return to in future (see the stack update sketch in Figure 3.3). Often this return pointer will allow us to make conclusion about what code made the call in the first place. For example if we had code such as

```
case f x of
  D y → e
```

Listing 5.4: Return Pointer Example

when calling `f x`, we would push a code pointer that allows us to resume computation of the `e` branch once we are done with that part of evaluation. Therefore in principle we can find any nouns associated with this call frame simply by walking the stack and identifying such return closures.

However, this technique is still quite inferior to full causality tracking, and even to evaluation scoping. The reason is that often there is no reason for the compiler to push a return code pointer. Just consider the following code:

```
g y = let x = f y
      in f x
```

Listing 5.5: No Return Pointer Example

Now we have two calls to `f`, but neither would cause a proper return closure to get generated. The first call will only result in a thunk update, which is something that happens so often that Haskell has prepared "canned" return pointers for this specific task. This means that all information about where the thunk originated from is effectively lost – at least, unless we are willing to play tricks with potentially left-over thunk pointers [Rouhani-Kalleh, 2014]. Yet the second call is even worse: Here the function is called in tail-call position, which means that the compiler will optimise any return code out and simply arrange for `f` to return to whatever called `g` in the first place. In this case, the whole stack frame is lost to us.

In the end, we will not pursue this approach any further at this point, as it would cause quite a bit of sampling complexity without actually providing any guaranteed benefit. However, it should be noted that this kind of data can be exceptionally useful in debugging, such as for locating program exceptions [Rouhani-Kalleh, 2014] or even tracking down crashes through the foreign language interface [Schröder, 2014].

```
data Expr b
  = Var    Id
  | Lit    Literal
  | App    (Expr b) (Arg b)
  | Lam    b (Expr b)
  | Let    (Bind b) (Expr b)
  | Case   (Expr b) b [(AltCon, [b], Expr b)]
  | Tick   (Tickish Id) (Expr b)
data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))]
```

Listing 5.6: Ticked Core Definition

5.4 Core

The last section introduced the design concepts behind our profiling solution. We now know what performance data we wish to collect and how we plan to explain it to the user. However, this still leaves quite a few concrete implementation details to address. After all, our goal is to integrate our profiling code into the GHC language tool chain, which is a formidable task as are dealing with a fully-fledged development environment.

This means that we have to account for a significant amount of related infrastructure. On one hand this means that we can count on a stable Haskell implementation with reliable existing frameworks for collecting, processing and analysing profiling-related data. Yet on the other hand, these solutions will be geared towards their specific use cases, so we will have to adapt them. This is especially tricky as we want to remain a "good citizen" in the sense that we do not want to disturb existing functionality.

This is especially pronounced for the compilation process, which we are going to start considering in this section. After all, this is where we want to introduce our nouns and track them all the way to the final compilation output using the causal model we introduced in Chapter 4. This is one concern where we are definitely not alone: The intention to annotate intermediate code with references to the original source code is something that we share with virtually every profiling-type solution in existence. Yet our annotation philosophy is unique enough that we will have to introduce a substantial number of special cases at the Core stage.

5.4.1 Tick Framework

In order to keep compiler software complexity low, it is therefore a good idea to generalise over annotations as much as possible. Fortunately, Marlow [2011] already recognised this issue and implemented a framework for Core annotations into GHC. This extension allows us to insert so-called "ticks" at arbitrary points in the Core syntax tree[3]. This

[3]The "tick" name probably derives from HPC ticks [Gill and Runciman, 2007].

means that relative to our first definition of Core in Listing 3.6 on page 28 we gain a new Tick constructor as shown in Listing 5.6.

While this generalisation makes introducing annotations as simple as defining a new constructor for Tickish, we have to acknowledge that not all annotations are born equal. In fact, depending on the intended use-case ticks might vary greatly in what information they carry and how they are expected to interact with code transformations. To integrate our annotations gracefully into the existing environment, we have to know the existing annotation types and their properties:

- The scoping ProfNote is used to maintain the cost-centre stack as used by cost-centre profiling [Sansom, 1994]. This can be used to map performance statistics such as execution time, allocation amount or heap residency to cost-centre stacks. We already shed some light on their scoping philosophy in Section 5.3.2 forward, which should have made it clear that retaining these through compilation poses comparatively strict requirements on code transformations.

- The counting ProfNote variation can be used to count entries to cost-centre stacks. These annotations still require cost-centre stacks to be kept consistent, but in practice we will see that they interact differently with respect to code transformations. Internally, GHC will distinguish both types using flags, but for the purpose of this work we will simply treat "scoping" and "counting" ProfNotes as distinct tick types.

- HpcTick works in a similar fashion, with the difference that it actually specialises on the task of coverage checking [Gill and Runciman, 2007]. In terms of code transformations, these annotations do not care about scoping, but want the compiler to ensure that they get invoked exactly if the associated code played a role in program evaluation.

- On the other hand, Breakpoint is a tick type that is purely used by GHCi, the read-eval-print loop of GHC. These annotations mark points in the program where we might stop program execution, for example for walking through evaluation step-by-step or inspecting variables [Himmelstrup, 2006; Iborra and Marlow, 2007]. It is fairly important that such breakpoints stay in place, therefore annotations generally restrict transformations heavily. However note that code for GHCi will barely get optimised in the first place, so in practice these restrictions rarely matter.

To introduce a new kind of Tickish into this mix means that we will have to figure out exactly where we fit into this continuum of annotation characteristics.

5.4.2 Source Notes

As our main goal is simply to annotate source code spans, we will be calling our source note type/constructor SourceNote. Here is the concrete definition:

```
data Tickish id
  = [...]
  | SourceNote { sourceSpan :: RealSrcSpan
               , sourceName :: String }
```

Listing 5.7: SourceNote definition

As introduced back in Section 2.2.2, our main goal with these annotations is to allow the user later to easily identify the piece of code that we are talking about. Whether this is more easily achieved by pointing out the concrete source code or referring to it using a name depends on the situation. Hence we retain enough information to allow us to use both identification methods.

Just like all other tick annotations, source notes will be generated by GHC's Coverage pass, which runs while the parsed Haskell program gets desugared into the intermediate Core representation. At this point it is quite straightforward to obtain the information mentioned above: Source spans for parsed program elements are automatically generated by the happy parser [Gill and Marlow, 1995] and we can easily retrieve them from the parse tree. Similarly, courtesy of automatic cost-centre generation there is an existing mechanism for automatically generating meaningful annotation names. The idea here is simply that we take all declarations that we find on the lexical scope, so for example a function g defined locally inside another function f would be assigned the name "f.g".

5.4.3 Semantics

While it is easy to define and generate source notes, we have to be a bit more careful about what source notes should actually *mean*. Clearly our idea is that we want them to model the annotations we introduced on the Core model in Chapter 4, but we still need to explain how exactly the mapping should look like. So let us now say that we have a source note src<a.hs> [4] as an instance of a cause term α, as well as a GHC Core expression e corresponding to an expression e in our Core representation.

Then we define the following terms as equivalent:

$$\mathsf{src}{<}\mathsf{a.hs}{>}\ \mathsf{e} \quad \Longleftrightarrow \quad \langle\!\langle \alpha \rangle\!\rangle e$$

using push annotations as defined in Section 4.8.2 on page 102.

[4]We will pretty-print ticks as tick<...>, but remain informal about what information they encode.

We are consciously *not* using simple annotations here. While this would allow us to represent causal dependencies more accurately, it would make it more complicated to reason about the meaning of our ticks. After all, using the push annotation we can now state that we see it as annotating all effects of code that comes below it in the syntax tree. A direct annotations would only do this provided the expression in question is not a `let` or one-branch `case` expression. However this means that for the purposes of annotated Core, there are now annotations that can not be represented in GHC's ticked Core. Suppose for example that we have two cause terms α and β, with α not being contained in β:

$$_{\langle\alpha\rangle}\texttt{let}\,\{f = ...\}\,\texttt{in}\ _{\langle\beta\rangle}\underline{e}$$

The only way to annotate the `let` expression in Core would be using one of our SourceNotes. But if we are limiting ourselves to push annotations, this means that we cannot prevent annotating e at the same time. Given that we assumed that β does not already contain α, this unfortunately means that our cause terms have become less expressive.

What we will be doing is the same slight dishonesty that Sansom [1994] also committed in the same situation: We will simply ignore the causes of the cost emitted by the `let` expression. Instead we can simply attribute this cost to whatever scope the `let` expressions ends up in. For the above example this would be β:

$$_{\langle\!\langle\beta\rangle\!\rangle}\texttt{let}\,\{f = {}_{\langle\!\langle\alpha\rangle\!\rangle}...\}\,\texttt{in}\ \underline{e}$$

Note that we did not eliminate α entirely, but moved it inside the binding instead. So we are essentially ignoring the causal history of the binding here, but retaining it for the bound *expression*, which is arguably the more important part. Another notable point here is that, after code transformations, the existence of the f binding at a certain point is more of a property of β anyway. After all, floating transformations will move or eliminate f depending largely on the make-up of $\underline{e}$. By this line of thinking we are not even over-annotating the binding costs by putting β on them, as $\underline{e}$ is essentially protecting the `let` expression from getting removed.

5.4.4 Annotation Combination

Another property of our abstract annotated Core language from Chapter 4 is that we were able to compose annotations freely. In fact, handling code transformations depended quite heavily on the ability to merge annotations where it made sense. However at this point we have only described SourceNote ticks in terms of adding a single primitive annotation to an expression at a time.

The obvious solution here is to "decompose" complex annotations into tick "stacks":

$$\text{src<a.hs> src<b.hs> e} \iff {}_{\langle\langle\alpha\beta\rangle\rangle}e$$

This lets us "compose" causes in a straight-forward way. Adding a new annotation corresponds directly to creating a suitable Tick node, not unlike the double-annotation notation we introduced back in Section 4.2.3 on page 63. Furthermore, we would ideally like to support various kinds of ticks to be able to co-exist within a Core tree, which means that we need machinery to traverse such nested ticks anyway. However note that representing compound causes as series of nested Tick expression is not quite optimal as far as compilation performance goes. After all, we are running the risk of bogging down compilation with re-creating deep tick trees every time the compiler wishes to change the contained expression.

To help prevent such deep trees from forming, it is important to minimise the amount of duplicated information we introduce. To be specific, note that the source code span referring to a Haskell expression can generally contain the source code spans of many other source code spans. Causally speaking, if parent source span contains some code, it directly implies that the child source span contains the given code as well. This makes the annotation redundant:

$$(\alpha \Rightarrow \beta) \Rightarrow (\alpha \wedge \beta = \alpha)$$

Consequently we never want to retain an annotation if it is made redundant by another on the same expression. This is quite a powerful mechanism, as it limits the annotations to disjoint pieces of the source code. In practice, Section 6.1.3 on page 182 will show that this allows us to use quite fine-grained SourceNote annotations without compilation time increasing significantly.

5.4.5 Scoping

Let us look more closely at how we want source notes to be treated during compilation. At this point we must remember that we are but one type of tick amongst several, and that each has its own unique set of properties and invariants. Where HPC ticks only care about whether or not the source code it refers to gets run, cost-centre profiling notes cares about the whole tree that they get placed on. We would ideally like to manage these complexities without the need to introduce separate special cases for every optimisation pass and every tick type. Therefore we generalise the tick properties as well by classifying them by how "fragile" they are against certain kinds of code transformations. Keeping the characteristics of existing tick types in mind, we end up with 3 fragility dimensions: "Scoping", "Placement" as well as "Counting" properties.

Let us begin with the "scoping" behaviour of ticks. This property says whether we care about whether the entirety of the annotated Core sub-tree stays within the annotation. The prime example here is the scoping ProfNote, which implements cost-centre profiling. As we introduced in Section 5.3.2, its aim is to ensure that the designated cost-centre becomes a part of the cost-centre-stack for all cost that gets generated within the annotated chunk of the program. In a similar way, SourceNote makes a causal statement about the causes for the contained code, and it would not make sense for code to be able to simply escape this scope. On the other extreme, HpcTick does not care at all about sub-expressions, as long as we guarantee that coverage stays virtually the same.

However, cost-centre-style scoping and SourceNote-like "soft" scoping still have substantially differences. To see why, consider an instance of inward `let` floating, which in GHC is implemented by the FloatIn pass. Such floating passes are the main way for GHC to relocate code pieces within the program, and it is important for our SourceNote to support it. Suppose that we want to float a let binding inside an arbitrary tick:

```
let foo = bar in tick <...> baz
   ⟹
tick <...> let foo = bar in baz
```

Listing 5.8: Floating Inwards

If tick was a ProfNote in this example, we would have to forbid this transformation. After all, this would enter new cost from bar into the context, which as we noted in Section 5.3.5 on page 126 would be invalid for cost-centre semantics[5]. On the other hand, this is perfectly valid according to our causality semantics – we established this in Section 4.8.6 on page 108. In fact, this transformation only ever changes the attribution of the let costs, which as mentioned in Section 5.4.3 we have decided to tolerate.

However, completely preventing code from floating inwards would be severely restricting, as it would block all kinds of code unfolding. Fortunately, this can be shown to be a special case with respect to cost-centre semantics. Suppose we float a lambda binding with the plan of in-lining and β-reduction:

```
let foo = λy → bar in tick <...> ... foo x ...
   ⟹
tick <...> ... (let foo = λy → bar in foo) x ...
   ⟹
tick <...> ... bar[x/y] ...
```

Listing 5.9: In-Lining

[5]Theoretically we could allow it if there was a matching `scc` annotation on `e1`. But this is not too likely, and therefore not implemented.

This particular transformation is actually trivially legal under evaluation scoping (see Section 5.3.3), as the application site determines what cost-centre stack to put the cost of bar in anyway. And after the β-reduction this holds true for the modified scoping proposed by Marlow [2012] as well, as the lexical scope of the call-site is more specific than the lexical scope of the definition site.

Let us now consider floating into the opposite direction. In the following example we are moving a binding towards the top level of the program:

```
tick <...> let foo = bar in baz
    ==>
let foo = tick <...> bar in tick <...> baz
```

Listing 5.10: Floating Outwards

This transformation actually happens at a number of places in GHC. For example when converting a constructor application to A-normal form [Flanagan et al., 1993] we want to `let`-bind certain sub-expressions, which might involve floating through a tick in the process. Happily, this time around we have a solid option for retaining lexical scoping properties: By wrapping bar into a copy of our tick, we ensure that the same code stays covered after we performed the transformation. Note however that we can not split every kind of tick this way, as we will see in Section 5.4.7.

To finish our overview of scoping behaviours, we have to mention that some types of ticks do not care about scoping at all. For example, HpcTicks would simply allow the following transformation:

```
tick <...> let foo = bar in baz
    ==>
let foo = bar in tick <...> baz
```

Listing 5.11: Floating Outwards Without Annotation

This means that we have identified three tick scoping behaviours:

- **cost-centre:** The most restrictive scoping policy. We expect lexical scopes of code to neither grow nor shrink. The only exception is where mixed cost-centre semantics by Marlow [2012] provably assigns the same cost-centre stacks.
- **soft**: A more permissive policy we intend to use for our SourceNote. Here we expect the lexical scope to only ever grow. Or put another way: Once we put an annotation on code – and therefore its costs – we expect it to still be there after optimisations have run their course.
- **none**: When the tick does not care about lexical scopes.

Figure 5.2 on the next page summarises the resulting tick property mapping.

	ProfNote (scoping)	ProfNote (counting)	HpcTick	Breakpoint	SourceNote
Scoping:	cost-centre	none	none	cost-centre	soft
Counting:	no	yes	yes	yes	no
Placement:	cost-centre	runtime	runtime	runtime	non-lambda

Figure 5.2: Tick Classification

5.4.6 Scoping Transformation Examples

While floating transformations are a good way to introduce the characteristics of scoping properties, optimisation behaviour will actually quite often depend on tick scoping. Let us first have a look at GHC's Simplify module, which groups together a large number of local code optimisations. The way that this pass is structured, we walk the program using a Zipper-like data structure [Huet, 1997]. For example, entering the function expression of an application expression $(e\ x)$ means that simplification recurse into e with a context of the form $(\Box\ x)$. This allows optimisations to react to their surroundings. For example if find that e is – or becomes – a lambda application, we can now look at the context and decide that fusing the two using β-reduction would be a good idea.

To introduce ticks into this process, we would ideally like to leave this continuation stack untouched. This would guarantee that all transformations see the exact same context that they would normally have, and therefore perform the exact same transformations. However, this means that we will have no way of retaining ticks inside expressions that get re-built from the context. As we do not want to lose ticks, we therefore must put them on top of the result expression, effectively floating all context expressions past them.

Fortunately, GHC only tracks context as far as the control flow is static, so for example we would not traverse out of a let binding or case branch. Therefore there are only a moderate number of instances where this would change things. To be precise, we would be performing the following transformations:

```
(tick <...> e) arg              ⟹ tick <...> (e arg)
f (tick <...> !a)               ⟹ tick <...> f !a
let !x = tick <...> e in ...    ⟹ tick <...> let !x = e in ...
let ... in tick <...> e         ⟹ tick <...> let ... in e
case (tick <...> e) of ...      ⟹ tick <...> case e of ...
case ... of ... → tick <...> e  ⟹ tick <...> case ... of ...
tick2 <...> (tick <...> e)      ⟹ tick <...> (tick2 <...> e)
```

Listing 5.12: Floating Ticks

The “bang” patterns on !x and !a show that strictness analysis could derive usage of the given binding to be strict. tick2 stands for another tick in the program, which might have different scoping properties and therefore needs to stay in place. Finally note that transformations (4) and (6) simply refer to let and one-branch case floating, which we have already covered in the last section[6].

Looking at these transformations with lexical scoping in mind, we can attest that every single one might move costs into the scope of the tick in question. Even pushing another tick inside has the potential to produce costs due to instrumentation code. This means that the tick in question must allow this, which is only the case for ticks that permit at least “soft” scoping, i.e. not ProfNote or Breakpoint. On the other hand, we can explicitly allow this for SourceNote. Doing these transformations has the nice side-effect of pulling annotations on the same control path together, which together with annotation elimination as explained in Section 5.4.4 can significantly reduce the amount of source annotations within the program.

Moving on, another GHC pass that needs to know about ticks is the occurrence analysis pass OccAnal. The purpose of this pass is to annotate binders with their usage information, which informs decisions about how greedy GHC should be about in-lining it. Consider the following example:

```
let x = e in tick <...> ... x ...
   ⟹
let x = e in tick <...> ... e ...
```

Listing 5.13: Thunk Unfolding

Here e is a thunk, which we are considering to unfold to a usage site. The conditions under which this is a good idea are not quite trivial [Peyton Jones and Marlow, 2002], but the general idea is that we want to check whether we could introduce redundant evaluations of e by changing the program in this way. Ticks complicate this matter further: Unfolding e would add more costs into the lexical scope of the tick, therefore we cannot allow this for ticks that have cost-centre scoping. On the other hand, we allow it both for SourceNote ticks and all other ticks with equally permissive scoping rules.

To close this section out, note that the only point where we distinguished between “soft” scope and “none” scope was when we decided whether to copy ticks on bindings when floating outwards. This means that between the two scoping classes, the only change will be how optimisations annotate their result. So far soft scoping implements exactly what we had in mind: Allow arbitrary transformations, simply by moving out of the way where appropriate.

[6]In the context of GHC both Simplify as well as FloatIn implement floating, but due the static control flow limitation this version is less powerful.

5.4.7 Counting

The second major tick characteristic is whether they care about entry counts. The motivating example here is coverage analysis using HpcTick: This tick checks whether the annotated expression gets evaluated during program execution.

The requirements work out "backwards" when compared with scoping: Whereas normally we care about the evaluation of the annotated code, now we care about how often outside evaluation causes our expression to get visited. This is actually more problematic, as now any change in sharing could potentially increase or decrease entry counts in completely unrelated parts of the program. For this reason we can only truly guarantee that we will never change the program so that the entry count of a "counting" annotation would fall to zero – or the other way around, for that matter.

However, we still make a best effort to retain realistic entry counts. For example, normally GHC would occasionally decide to inline "cheap" thunks, which would increase entry counts of any annotations. Moreover, note that "copying" an annotation on a floating let binding like in Listing 5.10 would also increase entry counts. On the other hand, floating outwards like in Listing 5.11 could be allowed. Incidentally, this demonstrates why ProfNote needs to be splittable into a scoping and a counting part, as this behaviour would be incompatible with cost-centre style scoping.

5.4.8 Floating Ticks

In the end, there are relatively few transformations where the counting tick property makes a difference on its own. However, there are actually a significant number of transformations that are invalid for *both* counting and cost-centre scoped ticks. This class of transformations is of special interest to us, because our SourceNote ticks will be the first tick type in GHC that has neither property, and can therefore support said optimisations. The crucial property is that we have maximum freedom with relocating ticks where necessary – as we introduced in Section 4.8.9 on page 112 for rules, "floating" ticks up is always an option according to our semantics.

A good example where we require this sort of freedom is η-reduction:

```
λx → tick <...> (e x)  ⟹  tick <...> e
λx → e (tick <...> x)  ⟹  tick <...> e
```

Listing 5.14: η-Reduction

Saving both a closure as well as an application is a big win, so we would really like to do this wherever we can spot it. Yet both versions would be invalid for either cost-centre scoping as well as counting ticks, as the relocation will change both evaluation scopes as well as entry counts. In contrast, for our causality semantics the first transformation is actually fine, as it would just correspond to adding more annotations on the closure value. On the other hand, moving annotations from the parameter to the application is clearly a change that could decrease profile accuracy. Unfortunately this is one of the situations where we have no other place to put the tick, and we will therefore have to accept the resulting annotation overreach. Note that ProfNote side-steps the second case by not putting ticks on variables, as we will see in Section 5.4.10.

We have a similar situation for Rules: As mentioned in Section 4.8.9 on page 112, this is all about supporting user-supplied transformation rules. As we can generally not derive where the right place would be to put the ticks afterwards, we have to be conservative and float all of them out of the way. For example, we would do the following for the short-cut fusion from Listing 3.4 on page 25:

```
foldr (tick<1> step) (tick<2> start) (tick<3> build g)
 ⟹ tick<1> tick<2> tick<3> foldr step start (build g)
 ⟹ tick<1> tick<2> tick<3> g step start
```

Listing 5.15: Rule Application

Relocating ticks like this is fairly easy, as rule applications let-bind and float non-trivial expressions where they match rule variables. We can use the same mechanism for "floating" our ticks. This has the nice side effect that the let in question automatically ends up with the appropriate ticks on it. For example, suppose that expr stands for a non-variable:

```
foldr (tick<1> expr) (tick<2> start) (tick<3> build g)
 ⟹ tick<1> let step = expr in tick<2> tick<3> g step start
```

Listing 5.16: Rule Application with Floating

Which in this instance retains exactly the intended ticks on expr. Floating ticks like this still has significant potential for annotation overreaching, both for the bindings as well as for the code built by the rule. However as noted in Section 4.8.9 on page 112 this is the only way we can support custom rules, and for the fusion rule it actually reflects to the underlying causal dependencies quite well.

5.4.9 Merge Transformations

However, we are still not quite done with possible optimisations. In this section we will get to the most problematic transformations. For our purpose, these are passes that merge code from different sources. The most obvious example for this is common sub-expression elimination in module CSE. The core problem is similar to rules: We want this pass to match expressions even where they differ in terms of our ticks, but what do we do from there?

There is only one place where we could possibly put annotations – the reference to the replaced expression:

```
let x = tick<1> x in ... f (tick<2> x) ...
⟹ let y = tick<1> f x in ... tick<2> (f x) ...
⟹ let y = tick<1> f x in ... tick<2> y ...
```

Listing 5.17: Common Sub-Expression Elimination

Again moving ticks out of the way makes things work with respect to our causality semantics. Yet thinking about this in terms of profiling should make us uneasy. After all, both steps in the above example reduce profile quality: First we push tick 2 up so it covers `f`, only to then replace the code by a reference to `y` – which according to the definition of `y` means that we associate all costs with tick 1.

Furthermore, ticks on thunk references are basically invisible to our profiling: If we evaluate `y` we would immediately jump to the definition of `y`. After that point, the only way sampling code could possibly derive that we were coming from a usage site annotated with tick 2 would be by stack tracing (see Section 5.3.6), which we do not currently support. Therefore the ultimate effect of this transformation might very well be that the replaced expression effectively "vanishes" from the profile, leaving all cost with tick 1. A very similar situation happens when we attempt to optimise `case` expressions. Consider the following situation:

```
case ... of ... → tick<1> e
            ... → tick<2> e
⟹ case ... of _ → tick<1> tick<2> e
```

Listing 5.18: Common `case` Branch Elimination

This is quite similar to common sub-expression elimination: Again we have two expressions that we wish to merge. The difference is that this time we merge both ways, meaning that we are forced to merge all annotations in order to preserve the ticks' cost association.

A further example would be `case` branch elimination:

```
case x of
  ... → tick<1> x
  ... → tick<2> x
⟹ tick<1> tick<2> x
```

Listing 5.19: Identity `case` Branch Elimination

Here all case branches simply return the scrutinised value, so there is no reason to branch in the first place. Again we must annotate in a way that will render profiling unable to detect which branch has been taken.

In the end, it is pretty clear that there is simply no way to reduce code complexity in these examples without also losing information for profiling. In fact, we could probably argue that the optimisations shown in this section are probably rare enough that we might actually be willing to sacrifice them for getting better profiling results. However, it is still important to know that our profiling solution can in fact work even here while still maintaining at least a certain measure of correctness[7].

5.4.10 Placement

The final property of ticks mentioned in Figure 5.2 was "Placement". Where scoping and counting properties speak about general policies for contained code, placement is about special *exceptions* to these rules. After all, depending on the semantics of the ticks in question, we might have situations where it makes no difference whether we put the tick in one place or the other. Especially where it simplifies reasoning about the code in question, we should therefore aim to establish a certain consistency.

For example, all annotations have in common that they only care about what happens at program execution time. However, as we noted briefly in Section 3.3.2 on page 28, GHC's version of Core implements a variant of System F [Girard et al., 1989], where the program may refer to purely compile-time entities such as types. This construction ensures that it is easy to check type-correctness of the intermediate code, but will be stripped before we get to later stages of code generation. Consequently, placing ticks on such type applications, type lambdas or coercions makes no sense, and we consequently float ticks through them until we hit an expression that has runtime semantics. We refer to this as "runtime" tick placement.

Furthermore, for our SourceNote ticks we will decide to float ticks through all lambda expressions, moving the tick into the lambda body. The motivation for this is rather mundane: It turns out that for supporting arbitrary optimisations, ticks on lambdas get in the way so often that it is just not worth keeping them there. Fortunately it is not too hard to justify this. GHC will let-bind all lambda expressions before code generation, which means that we can think of this as transforming a lambda expressions, as follows:

```
tick <...> (λx → expr)
  ⟹  tick <...> let y = λx → expr in y
  ⟹  let y = λx → (tick <...> expr) in y
```

Listing 5.20: let-Bound Lambda Expression

[7]This also facilitates implementation. After all, just a few missing transformations can have a significant impact on the produced code. Therefore it is easier to shoot for complete synchronisation between optimisations for ticked and un-ticked code than for any partial solution.

The second step only changes cost association in terms of let cost as well as one literal. We especially would not want to let-bind an expression like tick<..> λx →expr, as we could only interpret this as a thunk producing a lambda! We conclude that floating source notes through lambdas is a relatively minor problem.

Finally, while cost-centre scoping is fairly restrictive, it actually has the most permissive placement behaviour. The reason is that we can often show that certain tick relocations leave the overall consistency of the cost-centre profile intact. This holds not only for lambda expressions, but also for a number of other examples:

```
let x = scc<...> C (scc<...> y) (scc<...> 3) in ...
  ⟹  let x = C y 3 in ...
```

Listing 5.21: Cost-Centre Placement

Just like with lambda expressions, both a constructor as well as a literal will correspond to a simple allocation, so there is no "code" there to track the cost for. On the other hand, variable y could at maximum be a thunk. As thunk execution would not care about the cost-centre stack at the evaluation site, this means that we can eliminate the annotation without significant changes to the cost-centre profile.

5.4.11 Example

To see how this works out in practice, let us return to our running example from Section 3.4 on page 30, the factorial function. If we run Core optimisations with annotations as explained, GHC will arrive at the following annotated version:

```
$wfac = λto# →
    src<fac.hs:15> src<Enum.lhs:500>
    let wgo = λn# →
          src<Enum.lhs:534> src<Num.lhs:89>
          case (==#) n# to# of
            True →
              src<Enum.lhs:535> src<Num.lhs:89> n#
            False →
              src<Enum.lhs:536>
              case $wgo ((+#) w_s2L5 1) of
                r# → src<Num.lhs:89> (*#) n# r#
    in src<Enum.lhs:532> wgo 1#
```

Listing 5.22: Core Result

For brevity we have simplified treatment of Bools slightly, and note that compared with Listing 3.24 on page 35 GHC has short-cut the last loop iteration to return $n_\#$ directly.

However, at this point we are more interested in the annotations that the Core optimisations produced: First note that unfolding annotated library code has resulted in a good number of low-level source code links. This is expected and generally a good thing, as it might help us in tracking low-level performance problems. On the other hand, note that we end up with just a single reference to the original program (`fac.hs`). It is not hard to imagine what happened: We know from Section 3.4.1 on page 30 that this code is the product of repeated rule applications, which as we have just explained will cause source annotations to "float" towards the function entry point. Furthermore, as explained in Section 5.4.4 we combine annotations where appropriate, which apparently eliminated all but one tick here.

5.5 Cmm

Most interesting optimisations for GHC happen at the Core as functional code is particularly suitable for applying aggressive program transformations. Once this optimisation potential has been exploited, GHC will transform the code into a more restricted imperative representation. This allows for more local transformations to prepare the code for back-end code generation.

The intermediate language used for this is Cmm [Ramsey et al., 2010], which just like its cousin `C--` [Peyton Jones et al., 1998] is a portable assembly language. The main idea is that we now strip high-level data such as type information from the code and start reasoning in terms of instructions, registers and pointers. While doing this, we leave the language just abstract enough that we can later map it to any back-end configuration we choose. This allows us to solve common low-level code generation problems such as data layout, register allocation or runtime system interface in a general way.

However, when we last mentioned Cmm in Section 3.3.3 on page 29, we skipped an in-depth look at these transformations. Our reasoning was that for the purpose of causality analysis, the far-reaching Core optimisations were the more interesting target for our analysis. After all, in contrast to what we considered at the Core stage, most Cmm transformations will barely change the structure of the generated code, therefore leaving any annotations trivially correct.

However annotation implementation still faces a unique problem: Both cost-centre annotations as well as coverage ticks will get translated into Cmm code at this point. This means that there is actually no precedent for how to treat annotations at this level. Unfortunately, translation to Cmm code is no option for source code links, as we want to work without instrumentation code. This means that we have no choice but to design an entirely new solution for maintaining annotations on the Cmm level.

5.5.1 Cmm Example

To get a feel for what this will look like, let us now translate the Core code from Listing 5.22 into Cmm. If we simply ignore annotations, this would result in the following code for the worker function wgo:

```
$wgo()
{ A:  if (Sp-16 < SpLim) goto B; else goto C;
  B:  call stg_gc_fun(R2, R1);
  C:  if (R2 == I64[R1 + 7]) goto D; else goto E;
  D:  R1 = R2;
      call P64[Sp](R1);
  E:  I64[Sp - 16] = F;
      I64[Sp - 8] = R2;
      R2 = R2 + 1;
      Sp = Sp - 16;
      call $wgo(R2,R1) returns to F;
  F:  R1 = I64[Sp + 8] * R1;
      Sp = Sp + 16;
      call P64[Sp](R1);
}
```

Listing 5.23: Cmm Example

We can spot immediately that the Cmm version looks very different from the Core code. Instead of nested expressions, Cmm procedures consist of a number of blocks, with each block in turn containing a series of instructions. Cmm control flow is explicit, using instructions such as goto or call to steer execution.

Furthermore, at this point the implementation of our factorial function has become a lot more explicit: Blocks A and B start by checking for a stack overflow. Blocks C and D then handle the case that we are done with calculating the factorial, in which case we return to the location given by P64[Sp], passing R1 as our result. Otherwise, block E implements the first part of the calculation: We push a new stack frame with return pointer F and a copy of the counter n# (residing in R2). Then we increment the counter and perform the recursive call. Once the call returns to block F, we will take the saved-back counter from before and multiply it with the returned product. Then we pop the stack frame and return to the code that called our function.

At this point it is not hard to convince ourselves of our earlier diagnosis: As every recursive call to this function will increase Sp, we will have $O(n)$ in stack allocation, which will probably cause us to call the stack overflow handler in block B quite often. This is quite likely to be a performance problem!

5.5.2 Introducing Ticks

In order to bring source code annotations into the fold, we need to find a good way to incorporate the ticks from Listing 5.22 into our Cmm code. Unfortunately, finding a good solution here is not entirely trivial. To get started, suppose that we simply generate a tick “pseudo instruction” every time we encounter a SourceNote in Core:

```
$wgo()
{ A:   if (Sp-16 < SpLim) goto B; else goto C;
  B:   call stg_gc_fun(R2, R1);
  C:   src<Num.lhs:89> src<Enum.lhs:534>
       if (R2 == I64[R1 + 7]) goto D; else goto E;
  D:   src<Num.lhs:89> src<Enum.lhs:535>
       R1 = R2;
       call P64[Sp](R1);
  E:   src<Enum.lhs:536>
       ...
       call $wgo(R2,R1) returns to F;
  F:   src<Num.lhs:89>
       ...
       call P64[Sp](R1);
}
```

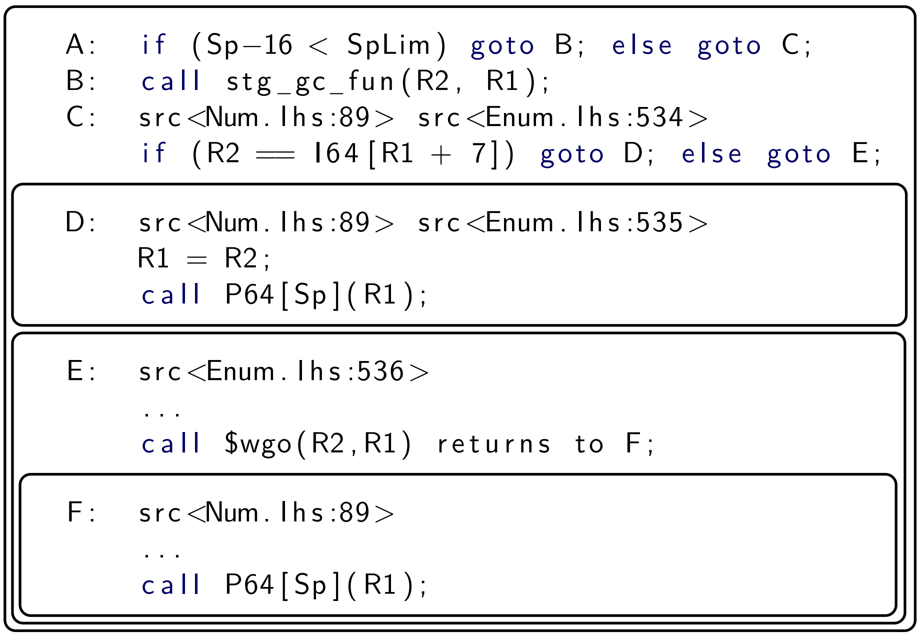
Listing 5.24: Cmm Example with Ticks

The good news is that this is a straight-forward translation, and the result seems to make sense for local annotations. For instance, the references to Num.lhs tells us that block D and F together perform a multiplication. Finding these source code links might already be useful for performance analysis.

On the other hand, we can also spot a number of problems. The entry code is not annotated at all, and the whole procedure contains no reference to the original source code whatsoever. If this was a larger program, this would make it basically impossible to tell what code we are looking at. The reason for this is that in Listing 5.22 we only have one source code annotation at the entry point of the function, which is basically out of scope for this procedure. This is a severe problem: Unless we can find a way to reconnect our ticks with their associated code, they will lose most of their usefulness.

5.5.3 Tick Scopes

The main disconnect stems from the fact that Cmm code consists of a loose collection of blocks, whereas Core code is naturally structured. As explained in Section 5.3.4 on page 125 it is vital that we can discover our static context, but the change in program



```
A:   if (Sp-16 < SpLim) goto B; else goto C;
B:   call stg_gc_fun(R2, R1);
C:   src<Num.lhs:89> src<Enum.lhs:534>
     if (R2 == I64[R1 + 7]) goto D; else goto E;

D:   src<Num.lhs:89> src<Enum.lhs:535>
     R1 = R2;
     call P64[Sp](R1);

E:   src<Enum.lhs:536>
     ...
     call $wgo(R2,R1) returns to F;

F:   src<Num.lhs:89>
     ...
     call P64[Sp](R1);
```

Figure 5.3: Cmm Tick Scopes

representation has made this a lot harder. After all, in Core we just had to walk the expression tree upwards towards the root, whereas now we lack such a robust navigation tool[8]. This leaves basically two ways to approach this problem: We could eliminate the need for code context by copying the complete tick context into every single block. However this would mean a massive amount of information duplication.

Therefore we will instead reimpose Core-like scoping behaviour on top of Cmm blocks. Figure 5.3 shows what this would look like for our example: The upper-most scope corresponds to the top level of the function `$wgo`, whereas the scopes starting from D and E implement the two main `case` branches respectively. Finally, the F block corresponds to the return code after the recursive call to `$wgo`. Maintaining this structure makes it easy for us to identify which ticks apply to which block.

When considering a suitable representation for these scoping rules, we have to keep a number of secondary factors in mind. Firstly, we want scopes to work even across Cmm procedures. After all, to resolve the issue of the missing reference to the original source file, we would need to imagine a parent scope containing all other scopes in Figure 5.3. This scope could then establish the causal connection to `fac.hs`. Furthermore, we have to remember that there will be optimisations working on the Cmm code. Keeping in line with our philosophy, we would like to be minimally disruptive to other compiler

[8]We could try to analyse the control flow of the Cmm procedure and e.g. take over ticks from dominating blocks [Lowry and Medlock, 1969]. While this would correspond nicely to our reasoning and might even be more accurate in some instances, it does not sound like a good idea architecturally to depend on control flow to conform to our expectations.

operations, both in terms of allowing transformations as well as keeping extra code complexity to a minimum. We not only want to be able to generate new sub-scopes on the fly, but also to merge them easily after the fact.

Finding a solution that satisfies such diverse requirements is not quite obvious. Our approach is that we represent a scope as a globally unique string, with every subset of this string corresponding to a potential super-scope. For our example we might assign the function `$wfac` from Listing 5.22 the scope string “S”. Then we could recursively produce new sub-scopes simply by appending the name of the first block of the scope[9], which would give us the following block coverage:

Scope	A	B	C	D	E	F
S	X	X	X	X	X	X
SA	X	X	X	X	X	X
SAD				X		
SAE					X	X
SAEF						X

Now we just modify Cmm code generation to associate every block with one of these scopes. This allows us to solve the present problem in an arguably elegant way: We know that a tick that gets generated into a block with scope SA is meant to apply to every other block in that scope as well as all sub-scopes. All blocks in `$wgo` would inherit any source note that we introduce into scope S, which would allow us to link it back to the original source code.

5.5.4 Optimisations

Furthermore, our scope representation allows Cmm optimisations to freely arrange blocks without having to spend too much work on updating our scope structure. For example, simply removing a block without ticks is always allowed. But there are even more involved optimisations that we can support. Consider for example concatenating two blocks together:

```
A:  ... stmts1 ...
    goto B;
B:  ... stmts2 ...
  ⟹
A:  ... stmts1 ...
    ... stmts2 ...
```

Listing 5.25: Cmm Block Concatenation

[9]This is just for easier presentation; GHC uses proper fresh names at this point.

This optimisation applies when there is no other jump to B, which is a situation that can happen quite often in generated Cmm code. As it can expose further optimisation opportunities, it is important that we allow code generation to take advantage of this. Consider what this means for ticks: Both stmts1 as well as stmts2 might be covered by a number of ticks from their respective scopes as well as all parent scopes. To ensure that the code remains covered we therefore assign the block a scope string that is the *union* of the old scope strings. Scopes of subsequent blocks are often similar, so this does not change much while ensuring proper tick scoping. So for example if we merged blocks with scopes S and SA, we could assure proper scoping by using the scope SA for the combined block.

We can do a similar transformation with common block elimination:

```
A:   ... stmts ...
B:   ... stmts ...
   ⟹
A:   ... stmts ...
```

Listing 5.26: Cmm Common Block Elimination

Under the assumption that A and B contain equivalent statements we can arrange control flow so we can eliminate one instance. However, note that this time around we have to take special care of tick annotations: Analogous to our treatment of common sub-expression elimination in Section 5.4.9 on page 139 we want to ignore the annotations when looking for candidates for CBE, only to re-annotate all ticks of the merged blocks on the final block. For scoping we would again use string union, which for block scopes SA and SB would yield us SAB, for example. This treatment allows us to maintain source code links throughout GHC's Cmm stage.

However note that we are actually glossing over a problem here: When assigning the merged block a scope string like SAB, we might actually reduce tick coverage for *other* blocks. After all, there might be another block in scope SA, which now has less coverage due to us moving the ticks into scope SAB! To solve this, we would have to assign individual ticks a scope – then moving them into new blocks would not change the scope they apply to.

On the other hand, to our knowledge this issue does not happen in practice, as GHC will only generate more than one block per scope if we have complex control flow. This generally means that the optimisations only get applied in situations where we eliminate complete tick scopes. For example, assume we want to use common block elimination on blocks from two scopes SA and SB. Then it is quite likely that CBE would have to merge *all* blocks from the two scopes at the same time, as jumps between them force equivalence.

5.6 Back-End

By this point in the compilation pipeline, GHC has transformed the original Haskell code to the point where we have a relatively low-level description of how the finished program should execute. The next step will be to produce actual machine code to be linked with the other components to form a complete executable Haskell program.

For our purposes this poses a fresh set of challenges: While so far we always had the option to change the language to fit our needs, we now have to target standardised file formats. This means that our options for storing profiling-related information is reduced to either shoehorning it into the given formats or separating it out and distributing it as a GHC-specific attachment.

We are going to use both approaches in this section: Where existing standard formats match our data reasonably well, we will provide it to the best of our ability. However we will also run into a few situations where we have no choice but to go separate ways and find a custom representation for our data.

5.6.1 DWARF

The trickiest part about this stage is that we want to keep track of machine code locations all the way until it is loaded into memory for execution. After all, this is what will allow us to identify the exact code under evaluation. We only have fairly indirect influence on this process: Once the Haskell program has been compiled to assembly code, it will be fed to standard assembly, linkage and loading tools just like a program of any other programming language. Predicting where these tools will allocate the object code is basically impossible unless we are willing to work with what these tools provide us. Fortunately, tracking object code for the purposes of debugging and profiling is not an entirely exotic use case. In fact, enough established language tool-chains have solved this problem that the involved formats have been standardised by the DWARF standards committee [DWA, 1993]. This specification defines a mostly language-agnostic way for associating debug data with the compiled code.

To remain flexible about exactly what debugging and profiling features to support, the DWARF format splits the data up depending on the concrete use case. These different aspects of the debugging data will then take the form of sections in the object file, with names such as `.debug_info` and `.debug_line` indicating both the format and the nature of the contained data. This will then allow the system linker to merge and update the sections accordingly when arranging object code for the final executable. Furthermore, we can count on being able to interpret this data for a running program, using standard tools and techniques in order to identify the concrete mapping of code locations to system memory.

5.6.2 Debugging Information

Let us first tackle the general debug information section .debug_info. This section is structured as a nested series of records, each with a number of fields as well as a number of sub-records. The top level record will always be "compile units", which stands for the result of compiling one source file. This would look as follows if we use objdump to pretty-print the data:

```
Abbrev Number: 1 (DW_TAG_compile_unit)
 DW_AT_name            : fac.hs
 DW_AT_producer        : The Glorious GHC System
 DW_AT_language        : 0x18      (Unknown: 18)
 DW_AT_comp_dir        : ~/leeds/test/haskell/fac/
 DW_AT_stmt_list       : 0x0
```

Listing 5.27: Compilation Unit Record

This introduces general data about the compilation, such as source file location as well as compiler and programming language. Note that the language ID 0x18 was assigned to Haskell on initiative of Howell [2012] and is slated to become part of the DWARF 5 standard, but is not yet recognised by standard tools.

Children of compile units will be subprograms, corresponding to Cmm procedures:

```
Abbrev Number: 2 (DW_TAG_subprogram)
 DW_AT_name            : fac
 DW_AT_MIPS_linkage_name: s3Sf_info
 DW_AT_low_pc          : 0x404360
 DW_AT_high_pc         : 0x4043b4
```

Listing 5.28: Sub-Program Record

This establishes just some very basic data that we have about the procedure: Its name as well as the program counter (PC) range we expect the object code to occupy. We will see in Section 5.6.4 how we derive the name from our source notes. Meanwhile, it is not hard to see how we can provide the latter information simply by referencing suitable code markers placed in the assembly code. For example, we would produce the above record by generating the following assembly code:

```
$wgo1_info:
    leaq -16(%rbp),%rax
    cmpq %r15,%rax
    jb _c3Tv
    // ... implementation ...
```

```
$wgo1_info_end:
```

Listing 5.29: Assembly With Bound Markers

Note the stack check, matching the start of the function in Listing 5.24. For debugging, the thing to note are the $wgo1_info and $wgo1_info_end markers, which allow us to refer to the function's bounds. This will ensure that standard linking tools will properly relocate code addresses in debug information whenever code changes location.

However, this is not quite enough information for us to represent the full address mapping that we had in mind. After all, at the end of last section we associated source ticks with individual *blocks*. Therefore we would ideally like to get the tool-chain to track block bounds as well. Unfortunately, this is the first point where we have to stretch the possibilities provided by the DWARF standard a bit, as there is no real equivalent for "block of code in an intermediate language".

Instead, we will declare Cmm blocks as named "lexical blocks":

```
Abbrev Number: 3 (DW_TAG_lexical_block)
 DW_AT_name          : c2Oj_entry
 DW_AT_low_pc        : 0x402ee9
 DW_AT_high_pc       : 0x402eef
```

Listing 5.30: Lexical Block Record

Even though the purpose of this structure is to provide variable scope, using it for assigning names to code portions works just as well. The name will allow us to associate additional information with each block later on.

5.6.3 Source Lines

While the records contained in .debug_info already would allow our tools to reconstruct the structure of the compiled object code, it lacks references to what concrete part of the source code the functions and blocks were coming from. In the spirit of the DWARF format, this is relegated to its own and largely independent section, namely .debug_line. One reason for this split is that line number information is normally not similar to other debugging information structurally. While C-like programs often have only a couple of procedures and lexical blocks, a complete program could consist of thousands of individual code lines. To support debugging, we require a full map connecting these to the compiled code in the object files.

As a result, there have been significant efforts to keep the encoding of this data as compact as possible. The DWARF standard DWA [1993] defines an elaborate data compression scheme, allowing the DWARF producer to customise the encoding for maximum efficiency. Fortunately, GHC can leave the generation of this table to the

```
    .file 1 "fac.hs"
.text
    // ... info table ...
$wgo1_info:
_c3Tc:
    .loc 1 15 1 /* test3 */
    leaq −16(%rbp),%rax
    cmpq %r15,%rax
    jb _c3Tv
.Lc3Tc_end:
_c3Tw:
    .loc 1 15 1 /* test3 */
    cmpq 7(%rbx),%r14
    je _c3Ts
.Lc3Tw_end:
    [...]
```

Listing 5.31: Assembly With Line Annotations

system's assembly tool. All we have to do is annotate the source code with special ". file " and ".loc" directives as shown in Listing 5.31. This will prompt the assembler to build the compressed file and line tables for us, as well as generate the .debug_line section. Here is the result, decoded using objdump:

```
File name        Line number     Starting address
fac.hs                    15             0x404360
fac.hs                    15             0x404369
fac.hs                    15             0x40436f
```

Listing 5.32: Source Line Table

Which is what we were hoping to see.

5.6.4 Source Note Selection

We have now seen how we can represent the information from a SourceNote in DWARF: We can use the name in order to refer to the procedure in .debug_info, and the information about the source code span to produce a matching entry in the .debug_line table. However, when we talked about tick scopes in Section 5.5.3 it should have become clear that we rarely want to think of blocks and source notes as being in a one-to-one correspondence. In fact, following static control flow backwards as explained in Section 5.3.4

on page 125, we will generally find many ticks to choose from.

Unfortunately, the DWARF format does not allow us to be indecisive about what source line or name we wish to associate with a piece of generated code. Therefore we are forced to pick: Which source note is the "most" useful for debugging the generated piece of code? We obviously want to throw away as little information as possible. This makes referencing, say, the multiplication operation a bad choice. After all, the program is most likely using multiplication often enough that it would not identify the code in question very well. On the other hand, knowing the current module function will generally allow us to discover roughly what low-level annotations we should expect.

This is why we will heuristically prefer source ticks that are from the source file that corresponds to the currently compiled module. The nice property of such source notes is that as long as we have implemented annotations correctly with respect to code transformation, it is virtually guaranteed that all generated code will be directly or indirectly annotated with a SourceNote corresponding to the original source file. After all, this was the case right after desugaring, and we would expect code to only gain annotations from there.

Bottom-line is that we will always chose the source tick that matches the compilation unit and is most relevant to the execution site according to Section 5.3.1. This has the additional benefit that the output also closely matches the expectations of debugging tools, as the source references now essentially allow "stepping" through the source file.

5.6.5 Unwinding

Providing source file information in DWARF format was our main goal, but we can actually improve on this further by taking advantage of more elaborate DWARF features. Remember that in Section 5.3.6 on page 128 we explained that we could recover more information about the current program state by walking the Haskell stack and analysing the return code pointers we encounter. This corresponds to the common debugging technique of stack unwinding: By successively removing stack frames from the heap we recover information about the call hierarchy.

And despite the fact that the Haskell stack is actually maintained in a fairly non-standard way, we can communicate the unroll procedure to the debugger using DWARF unwind instructions. For example, Listing 5.33 shows the unwind specification for the Cmm code from Listing 5.23. First, the Common Information Entry (CIE) establishes the general heap format. For us, this means that we expect the stack to grow downwards in steps of 64 bits, making the data alignment -8 bytes. Furthermore we establish how to unwind a stack frame: Set the instruction pointer to the value we find at the current Canonical Frame Address (CFA), and set the stack pointer rbp to the current CFA[10].

[10]Note that rbp is the default x86-64 back-end register used for the Sp Cmm register.

```
CIE
  Code alignment factor: 1
  Data alignment factor: -8
  Return address column: 16
  DW_CFA_offset: r16 (rip) at cfa+0
  DW_CFA_val_offset: r6 (rbp) at cfa+0
  DW_CFA_def_cfa: r6 (rbp) ofs 0

FDE cie=00000000 pc=0040435f..004043b4
  DW_CFA_set_loc: 00404397
  DW_CFA_def_cfa_offset: 16
  DW_CFA_set_loc: 004043aa
  DW_CFA_def_cfa_offset: 0
```

Listing 5.33: Unwind Instructions

By default, we set the frame base address CFA to the current rbp, which describes the situation at the point where a function is entered: Unwinding the stack would simply mean resetting the instruction pointer using the return pointer from the stack, leaving rbp unchanged. However, once a function does actual stack allocation, this has to be updated using a Frame Description Entry (FDE). In our example we instruct the unwind process that after the update of Sp in block E of Listing 5.23 we will have to roll the stack pointer back suitably.

Note however that there are subtleties to this. Firstly, typical debugging tools will actually decrease the return pointer by one byte before trying to look up its information. This is under the assumption that typical code would looks like follows:

```
call fun
... // return code
```

Listing 5.34: Expected Assembly

If executed, the call instruction in this assembly would push a return pointer pointing to the return code. However, for debugging we are actually interested in where execution was coming from, not where it is going to take us next. And in this specific case we can easily reconstruct that, simply by decreasing the instruction pointer. In the above code this clearly leads us back to the original call instruction.

Unfortunately, this small trick does not work for Haskell code, as we have no guarantee that the calling code can be found anywhere near the return pointer. Let us have a quick look at the recursive call in our running example:

```
        movq  ret_info,(%rbp)
        jmp $wgo_info
        .align 8
        .quad 65
        .quad 32
    ret_info:
        ... // return code
```

Listing 5.35: Actual Haskell Assembly

where the .quad directives describe the info table, which communicates the heap layout for GHC's garbage collector. This means that decreasing the return pointer will actually yield a non-code pointer! This is why we actually implement a subtle "hack" to get around this problem. Note that in Listing 5.33 the FDE actually points to code addresses that are shifted by exactly one byte. so instead of starting at our true function entry 0x404360, we are referring to 0x40435f instead.

In order to get good line number information for past stack frames, we have to make the same considerations for source lines. As explained in Section 5.6.3 we are not generating this table ourselves, so we have to shift the position of the .loc directive "manually" in order to produce the intended behaviour. Fortunately, we know that decreasing any return pointer will land us directly inside the info table, so we can simply annotate the data:

```
        ...
        .align 8
        .loc 4 89 19
        .quad 65
        .quad 32
    ret_info:
        ...
```

Listing 5.36: Line-Annotated Haskell Assembly

The easiest approach is to simply put the same source code annotation on the info table that we would have put on ret_info. However note that we could actually be smarter here: Just as with C code we might occasionally have information about where the control flow was coming from, therefore annotating the info table with a suitable line annotation could improve results.

However, there will always be examples where this is not possible, such as canned return targets. Here we actually have the additional problem that the return pointer is the start of the function. This means that decreasing that particular code pointer will actually go out of the function's bounds, both according to the debug information records as discussed in Section 5.6.2 as well as the symbol table. The latter is more critical, as we can not manipulate it due to its role in the linking process. Unfortunately, debugging tools such as gdb use it to lookup function names, leading to unhelpful ?? entries in stack traces. For example, the following stack trace would be generated for an exception raised inside a thunk update:

```
#0  stg_raisezh () at rts/Exception.cmm:433
#1  0x694330 in ?? () at rts/Updates.cmm:57
#2  0x4047a0 in $wfib_err () at stack-trace.hs:7
```

Listing 5.37: Example Stack Trace

Unfortunately, we cannot correct this from the compiler. A proper solution would most likely require a patch to the debugging tool in question.

5.6.6 GHC debug records

At this point we have pretty much exhausted what we can meaningfully express with the DWARF standard. However, we have a lot more information in store: In our mind every piece of object code is covered by a number of possible source annotations, each potentially providing valuable information about the nature of the code at hand. Furthermore, we might want to mix in additional information about the compilation process, as we will see in the next section.

In order to maintain such data alongside existing DWARF debugging information, we will allocate our own object file section, called .debug_ghc. For overall data organisation we will choose the event log format [Jones et al., 2009]. This is in anticipation for collected profiling information getting appended later on. Note that we cannot rely on debugging tools to know about the format of our section, which means that we will only be able to identify code addresses by comparing against information from the .debug_info section. This is why we took great care in Section 5.6.2 to associate a unique name with every procedure and block.

As we have seen in Figure 5.3 on page 146 tick scopes naturally form a tree structure, so it is a straight-forward choice to use a block tree for our debug records as well. Note though that scopes will not always directly conform to this, as shown in Figure 5.4. Firstly we might encounter multiple blocks sharing the same scope, as happens for AB in the example. We then choose a "representative" for the scope[11], which will carry all ticks as well as becoming parent to all other blocks covered by the scope. Furthermore, we might have scopes with more than one parent scope, such as ABC in the example. Then we have to settle for just one parent, and restore all "lost" ticks by replicating

[11]The choice is pretty arbitrary – to reduce randomness we select the first block according to pre-order traversal. This allows us to guarantee that the entry block will become root of its scope, if nothing else.

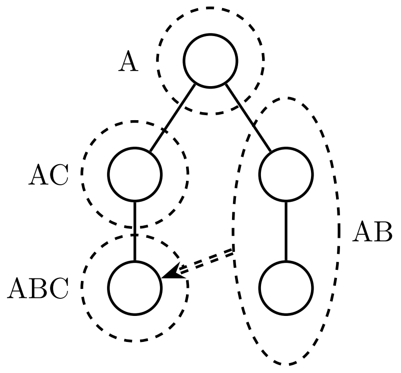
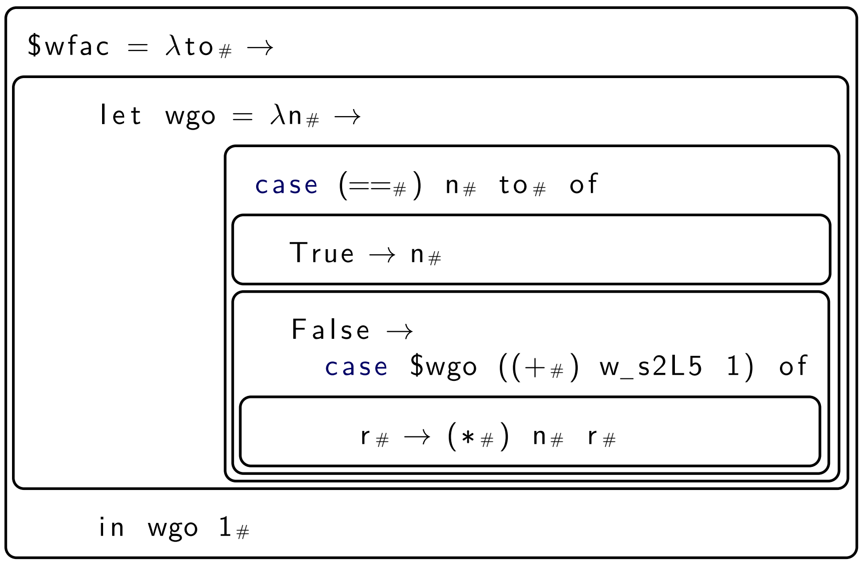


Figure 5.4: Block Tree from Scopes

```
Debug module main:fac.hs
Debug block label c3TG_entry id 1 parent 0
Debug block label c3TH_entry id 2 parent 0
Debug block label c3Tw_entry id 5 parent 4
Debug block label c3Tv_entry id 6 parent 4
Debug block label c3Tj_info id 8 parent 7
Debug source * (Num.lhs:89)
Debug block label c3Tt_entry id 7 parent 4
Debug source eftIntFB.go (Enum.lhs:536)
Debug block label c3Ts_entry id 9 parent 4
Debug source * (Num.lhs:89)
Debug source eftIntFB.go (Enum.lhs:535)
Debug block label s3Sf_info id 4 parent 3
Debug source * (Num.lhs:89)
Debug source eftIntFB (Enum.lhs:534)
Debug block label c3TD_entry id 3 parent 0
Debug source eftIntFB (Enum.lhs:532)
Debug block label c3TC_entry id 10 parent 0
Debug source eftIntFB (Enum.lhs:531)
Debug block label Main_zdwfac_info id 0
Debug source eftIntFB (Enum.lhs:531)
Debug source enumFromTo (Enum.lhs:500)
Debug source fac (fac.hs:15)
```

Listing 5.38: Debug Records



```
$wfac = λto# →
    let wgo = λn# →
                case (==#) n# to# of
                  True → n#
                  False →
                    case $wgo ((+#) w_s2L5 1) of
                          r# → (*#) n# r#
    in wgo 1#
```

Figure 5.5: “Scopes” in Core

them in the appropriate block. For our sketch, this would mean copying the ticks associated with scope AB into the block contained in ABC.

Once we have obtained such a block tree, we emit a series of event messages that describe its structure and payload. Listing 5.38 on page 157 shows the result for our running example: The entry block Main_zdwfac_info with ID 0 (line 20) is the root of our tree, corresponding to the top level in Listing 5.22 on page 142. One level below we find the entry block of the $wgo with instrumentation ID 4 (line 13). Note that as shown in Listing 5.28 on page 150, GHC has assigned it the unique name s3Sf, which we can use to identify the correspond .debug_info record and therefore its code range. As events are written as a series, associating further properties with these blocks simply means emitting extra block property events right after the event declaring the corresponding block. For our example, the root node 0 will receive three source annotations from fac.hs and Enum.lhs respectively.

5.6.7 Core Notes

When we talked about what information we could infer from just a static context in Section 5.3.4, we noted that it will be useful to allow the user to trace compilation backwards in detail. After all, while explaining source code causation for a program state might require us to bring a large number of possibly relevant source locations into play, we can generally pin-point exactly where we found ourselves in terms of Cmm or optimised Core code. This offers the user a great deal of information, especially making them independent of heuristics such as the one discussed in Section 5.6.4.

To make use of this property, we want to link up a block tree like the one shown in Figure 5.4 with a Core program, such as the optimised Core we derived for factorial back in Listing 5.22 on page 142. After all, as Figure 5.5 shows we can easily map tick scopes back to the structure of the Core program. In order to track this association, we need to define yet another tick type just like we did in Listing 5.7 for source code links. This time however, we track Core:

```
[...]
| CoreNote { coreBind :: Var
           , coreNote :: ExprPtr Var }
```

Listing 5.39: Core Note

with the coreBind providing a unique identifier for the Core location, and the ExprPtr referencing the Core expression or binding in question.

The unique property of the CoreNote is that it will only get generated once we translate the intermediate Code into Cmm. So it is guaranteed not to exist on the Core stage, so there is no need to decide floating or placement properties like in Section 5.4 on page 129. In terms of treatment on the Cmm stage we simply expect that they will remain in their associated tick scopes, just like source notes. In the end, this yields us a straightforward map of tick scopes to one or multiple locations in the Core code.

In order to save the Core for later inspection we then encode the Core code in the event-log format. We can simply achieve this by inserting appropriate “Core” event messages with the messages the other block records. However, this process is not quite obvious: To reduce space usage we aim to never emit any single piece of core twice. Therefore we partition the full Core code into separate chunks, using references in order to assemble them back together.

In our example, this means that we would generate the following Core pieces:

```
<A>  (*#) n# r#
<B>  case $wgo ((+#) w_s2L5 1) of r# → <A>
<C>  n#
<D>  case (==#) n# to# of True → <C>; False → <B>
<E>  $wfac = let wgo = λn# → <D> in wgo 1#
```

Listing 5.40: Core Pieces

To further increase the usefulness, we can also add type information into the mix. Note however that retaining too much data at this point can quickly lead to the produced binary growing excessively. After all, if we go through with this we will not only save every single piece of optimised Core that the compiler produces, but also pretty much all types encountered in intermediate code.

The latter can be surprisingly problematic, as for example heavy usage of monad transformers [Espinosa, 1995] in the popular haskeline library at one point produced types that spanned about 20 000 lines when pretty-printed. This is why we took great care to be space-efficient, and will make Core tracking an optional feature.

5.6.8 LLVM

To this point we have described what debugging data we would like to ship with our binaries in order to link object code back to profiling and debugging data. We even used examples of what we would like the produced assembly code to look like. Yet we have to note that for GHC compilation, we will not always have full control over the produced assembly. While it is generally recommended to have GHC handle the production of the assembly code directly, there are cases where it is beneficial to leave this task to an external tool. For our purposes, this means that we have to deal with an additional layer of indirection.

The alternative we want to cover here is the LLVM backend [Terei and Chakravarty, 2010], which generates high-level assembly code to be compiled using the LLVM compiler infrastructure [Lattner and Adve, 2004]. This allows us to make use of its low-level optimisation facilities, which are much more advanced than the fairly basic transformations that GHC can perform on Cmm and assembly code. For our purposes, this means that we have to find a way to instruct LLVM to generate the suitable DWARF code. Fortunately, there is fairly direct support for this in the form of LLVM *meta data*. For example, generating the .debug_info records for the compilation unit as well as the $wgo function would look as shown in Listing 5.41. Note the similarity with the structure of .debug_info as discussed Section 5.6.2 on page 150: Using metadata links we define a tree-like structure, assigning every node a tag deriving from the record types from the .debug_info section. Note that while most of the meta data is comprised of literals and other meta data, @s1E8_info is actually a reference to the function of the same name, which we supposedly defined earlier in the same file. This allows us to link the debug information to the definition in question.

Generating line number annotations also has parallels to how we generated the line table by inserting .loc directives into the assembly in Section 5.6.3. For LLVM, we simply annotate the code with suitable metadata links as shown in Listing 5.42. Notice the !dbg annotations, which connect LLVM code lines with the source code line in question. For example, meta data node !4 here stands for line 15 column 17 in fac.hs. This is also where we introduce lexical blocks, which serve the same purpose they did previously: Introduce unique names for Cmm blocks.

In the end, while LLVM allows us to get good results with relatively little effort, there is also a lot that we simply cannot express using LLVM's intermediate representation.

```
!0 = metadata !{ metadata !"fac.cpp",
                 metadata !"~/leeds/test/haskell/fac/" }
!1 = metadata !{ i32 786473, // DW_TAG_file_type
                 metadata !0 }
!2 = metadata !{ i32 786449, // DW_TAG_compile_unit
                 metadata !0, i32 24,
                 metadata !"The␣Glorious␣GHC␣System" ,...}
!3 = metadata !{ i32 786478, // DW_TAG_subprogram
                 metadata !0, metadata !1,
                 metadata !"$wgo", metadata !"$wgo",
                 metadata !"s1E8_info",
                 ...,
                 void (...)* @s1E8_info }
```

Listing 5.41: LLVM Debug Meta Data

```
define internal cc10 void @s1E8_info(...) align 8 {
c1Fk:
  %ln1Gp = add i64 %R1_Arg, 7, !dbg !4
  %ln1Gr = inttoptr i64 %ln1Gp to i64*, !dbg !4
   [...]
}
[...]
!4 = metadata !{i32 15, i32 17, metadata !5, ... }
!5 = metadata !{i32 786443, // DW_TAG_lexical_block
                metadata !0, ...,
                metadata !1, ... }
```

Listing 5.42: LLVM Line Meta Data

Most prominently, LLVM expects to handle its own stack, so there are no facilities for explaining how to unwind the custom GHC-style stack. Furthermore, there is no direct way of assigning names to lexical blocks like we did in Section 5.6.3, so to retain links to Cmm code we can only mark them indirectly, for example by creating dummy variables. This means that while profiling and debugging using the LLVM backend is possible, we have to accept further downsides on top of DWARF's inherent restrictions.

5.7 Data Collection

We have now obtained a complete debug information map for our object code. This means that at this point a single object-file code pointer would be enough for us to be able to tell the full story of how the code in question was produced, and especially what parts of the original source code played a role. All we need now in order to start our analysis is some performance data that yields us such code pointers.

In this section we will show how to collect such profiling data. Conceptually, this means that we are now leaving the part of the implementation that is primarily concerned with the "cause" side (source code) and enter the phase where we are looking for "effects" (resource consumption). Therefore this section will be about implementing the verbs and metrics introduced back in Section 5.2 on page 117. However, we still have to make sure that we carry along all data we have collected so far about nouns and explanations, as this will later allow the analysis tools to link up the two sides of the profiling problem.

5.7.1 Event-Log

As mentioned in Section 5.6.6 on page 156, we will be using the event-log format as defined by Jones et al. [2009]. Using a common standard format for profiling has the advantage that we can integrate various approaches both on the producer as well as the consumer side: Existing profiling infrastructure can provide further performance data to help put ours into context, and other profiling tools might decide to also take advantage of the data we collect. For example, apart from the original ThreadScope application the format is also understood by the Eden trace viewer [Berthold and Loogen, 2007], HdpHProf [Al Saeed et al., 2012] as well as the `ghc-events-analyze` tool [de Vries and Coutts, 2014], all covering different use-case for profiling analysis.

In order to produce a useful event-log file, we will first write out a "prefix" consisting of the program's debug records as defined in Section 5.6.6 on page 156. As we already generated these records in event log format, this mostly means copying data from our own `.debug_ghc` section into the fresh event-log file. However, this task is not entirely straightforward, as we want to provide debug information for all involved libraries as well, for which we have to resolve both compile-time as well as run-time relocations.

This means that the runtime system needs to be able to read the .debug_info section of both its "own" executable as well as all dynamically linked libraries. Fortunately, there are standard libraries available for this task, so we simply use `libelf` [Riepe, 2009] and `libdwarf` [Anderson, 2011] in order to extract all required information. We furthermore relocate all code pointers using our process' memory map[12] which yields us the actual address ranges for every code block with a DWARF record. By interleaving these pointer ranges with the existing debug block records, we obtain a full mapping of debug information to pointer ranges:

```
Debug module main:fac.hs
Debug procedure label c3TG_entry instr 1 parent 0
Debug pointer range 0x004043dd-0x004043e3
Debug procedure label c3TH_entry instr 2 parent 0
Debug pointer range 0x0040440c-0x00404420
Debug procedure label c3Tw_entry instr 5 parent 4
Debug pointer range 0x00404369-0x0040436f
Debug procedure label c3Tv_entry instr 6 parent 4
Debug pointer range 0x004043b0-0x004043b4
Debug procedure label c3Tj_info instr 8 parent 7
Debug source * (libraries/base/GHC/Num.lhs:89:19-89:30)
```

Listing 5.43: Debug Records with Pointer Ranges

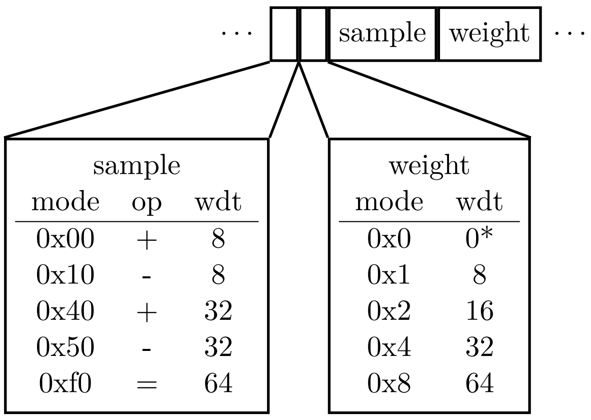
Where we have more information on source ranges than we have records in .debug_ghc, we can even synthesise new records that describe them. This might happen for example when we are linking with C libraries:

```
Debug module :SYMTAB: /lib64/libpthread-2.12.so
Debug procedure label pthread_cond_wait@GLIBC_2.2.5
Debug pointer range 0x3aa3c0c070-0x3aa3c0c108
Debug procedure label pthread_setaffinity_np@@GLIBC_2.3.4
Debug pointer range 0x3aa3c0fac0-0x3aa3c0fb65
```

Listing 5.44: Debug Records for C Library

Note however that this is limited to information extracted from the symbol table and the .debug_info section. Therefore instead of complete source references we end up with just symbol names. Extending this to cover full DWARF information would require us to decode the information in .debug_line as well, which we currently skip.

[12] Which we currently obtain by reading `/proc/self/map`, basically abandoning all hope of portability.



sample		
mode	op	wdt
0x00	+	8
0x10	-	8
0x40	+	32
0x50	-	32
0xf0	=	64

weight	
mode	wdt
0x0	0*
0x1	8
0x2	16
0x4	32
0x8	64

Figure 5.6: Sample Encoding

5.7.2 Samples

With the debug map established, referencing debug data becomes as easy as saving a pointer, leaving the actual lookup to the analysis tool. As Section 5.2.1 on page 118 explained this is good news, as low overhead is the easiest way to prevent skews.

On the other hand, putting some effort into our encoding scheme is a good idea anyway. Remember that this is about providing evidence for a connection between certain causes (nouns) and effects (verb). As explained in Section 2.2.4 on page 12 this is not just about a one-to-one connection, but we will want to associate weights with connections, corresponding to the approximate strength of the link. Depending on our collection method, the value range of these weights might vary quite a bit. For example, sampling only the instruction pointer we might only ever get one sample at a time, while inventory-type samples such as heap residency profiles will produce much more data (see Figure 6.12 on page 190). If we add the fact that our noun representation will generally be a 64 bit code pointer, it becomes clear that we might quickly be looking at about 16 bytes per sample, which could add up quickly unless we invest into some basic pre-processing.

To help with this, we use a basic compression system shown in Figure 5.6: A mode byte tells us for every weighted sample how both the sample value as well as the weight is going to be encoded. For values, we exploit that subsequent samples are often expected to be in close proximity to each other, making it more compact to emit a distance to the last sample instead of the full 64 bit pointer. We want to be especially mindful of small hot-spots, where samples are often in very close proximity (< 256 bytes), and unless samples jump between libraries pointer differences will generally fit into a 32 bit integer. On the other hand, for weights we have to account for the full range of values, with no expected proximity between neighbouring values. Therefore we cover the whole range of integers from 0 up to 8 bytes, with 0 standing for the constant weight of 1.

Furthermore, we will allow sample events to specify what noun and verb they are talking about. This will make our sampling infrastructure open for future extensions, for example using cost-centre nouns instead of instructions pointers. For the moment, here is how an example sample event would look like when carrying instruction pointers, collected by processor cycles:

```
Sample SampleInstrPtr by SampleByCycle cap 0:
  2*0x404fe9 , 0x404ff0 , 2*0x404fe9 , 3*0x404ff0 ,
  0x404fe9 , 4*0x404ff0 , 0x404fe9 , 2*0x404ff0 ,
  0x404ff0 , 0x405074
```

Listing 5.45: Samples in Event-Log

Especially note the distinct "hotspot" behaviour here: In this example most samples are from a region that spans just two bytes! We also observe that multiple samples have a weight greater than 1, as shown by the `n*` prefixes, even though the underlying sampling process actually produces samples individually. The explanation is that our encoding algorithm actually coalesces subsequent samples with the same sample value, exploiting another easy compression opportunity.

5.7.3 Timers

Now that the sample tracing infrastructure is in place, we just need to acquire actual sampling data in order to produce proper Haskell program profiles. Namely we want an approximate map of where certain effects (verbs) occurred. The most direct way to do this is sampling: We take snap-shots of the execution state at frequencies that conform roughly to the amount of (abstract) resource usage.

Just consider our primitive verb from Section 5.2.2 on page 118: Time passing. Finding program states where most time is spent is as simple as interrupting the program at set intervals, each time taking a sample of the current instruction pointer. Clearly a program state that only appears briefly will have a very low chance of appearing in our profile. On the flip-side, with enough data collection we can also expect program states to feature more often the more time was wasted.

On paper implementation is also relatively straightforward: All operating systems offer some sort of timer support. Often we can even obtain data about the thread state on these occasions. For example, we can use setitimer and sigaction on typical Unix systems to program a timer to pass us the thread's "context" in certain intervals, including the full register set. On Windows systems, we can might be able to achieve a similar effect by using SuspendThread followed by GetThreadContext. Access to registers means that we can sample instruction pointers and generate the sample messages explained in the last section, making "time" the first verb that we can provide profiling support for. Note that the register set will normally also include the current stack pointer – this could allow us to do stack tracing as explained in Section 5.3.6, potentially providing more expressive nouns[13].

In practice however, working with timers can be a bit of a challenge due the fact that the operating system interfaces are not strictly meant to be used for profiling. This means that we will often have to help things along a bit in order to get exactly what we want. For example, `setitimer` will only ever raise the signal for one thread, therefore we opt to manually replicate the signal across our program for multi-threaded applications. Such tricks are not only prone to bugs, but also tend to result in bad performance.

5.7.4 Hardware Performance Counters

Profiling by time is rather simplistic; as explained in Section 5.2.3 we might want to unpack program performance further by focusing on the different influences on CPU execution speed. Fortunately, this is again a concern that we share with a lot of existing profiling tools. And in fact the art of collecting specialised profiling information has matured to the point where we even have hardware support in the form of hardware performance counters. The idea is that we can program these counters to track some kind of low-level performance metric such as processor cycles or cache misses. This will give the operating system a notification whenever a certain threshold is reached. Given that modern processors support dozens of possible counter configurations, this gives us a lot of flexibility for data collection.

The principal “standard” library for programming hardware performance counters is the Performance API, or PAPI for short [Browne et al., 2000]. It is portable across a number of different Unix systems, and allows us to define “overflow” handlers for a wide variety of performance counters from cache statistics over branching data to simple instruction counts. However, performance is limited due to the fact that it needs to call the program every time to allow taking samples. Furthermore, support on modern machines does not seem to be completely stable [14]. Fortunately, most modern operating systems offer better alternatives.

5.7.5 Perf-Events

Linux is a popular kernel for Unix systems. Where it is present we can obtain performance data much more directly by talking directly to the operating system. The idea is that the `perf_event` kernel interface [Eranian, 2009] allows us to instruct the kernel to take over the task of collecting instruction pointer samples entirely. The collected

[13]However a possible implementation would have to ensure that we are in valid Haskell code to begin with, or the sampling code might end up crashing the program!

[14]Based on the episodic evidence that it mysteriously stopped working on the author’s computer.

data will then "stream" back to the process using a memory map, which reduces our job to copying it into our event log in regular intervals.

The advantage of this approach is that we have to accept a lot less overhead than before. After all, performance counter events have to be handled by the operating system at some level anyway, so having sample collection happen without another context switch is a significant win. On the other hand, we also have to accept less influence over the collection process. For example, now we do not have the option to walk the Haskell stack – while the Linux kernel supports a limited form of stack tracing, it is quite unlikely that it will be able to make sense of the Haskell stack anytime soon [15].

5.7.6 Allocation

Going beyond simple hardware statistics, Section 5.2.4 argued that we also have an active interest in more Haskell-specific aspects of program performance. One very basic characteristic of code generated by Haskell is that it often involves allocation of various types of memory. In the Cmm example in Listing 5.23 on page 144 we actually saw this spelled out: Whenever stack allocation reached a certain threshold, the generated code would call into the run-time system in order to request a fresh block of memory. Such requests are an indirect sign of either heap or stack allocation, both verbs that we would like to track. After all, in either case we are making a call into the memory subsystem, which is already quite costly on its own. Therefore the locations of allocation can be very interesting for performance analysis.

Therefore, instrumenting these calls into the runtime system is an obvious choice. Unfortunately, while the runtime routine receives a continuation code pointer, this pointer does not always directly refer to where the allocation took place. This means that if we have code like the following:

```
case x of
  C → ... −− lots of heap allocation
  D → ... −− little heap allocation
```

Listing 5.46: Ambiguous Heap Allocation Location

the heap check might assign the same return pointer to both of them, which means that we cannot tell which branch was actually causing the heap allocation in the first place. Unfortunately, this happens quite frequently with GHC-generated code.

[15]The kernel would essentially have to track down, relocate and use the unwind information we generated in Section 5.6.5 – which would be quite a bit of complexity even for a monolithic kernel.

5.7.7 Residency

Apart from allocation we also want to keep track of how much memory actually survives garbage collections. As explained in Section 5.2.5 this can have an indirect effect on program performance by reducing locality and increasing garbage-collection time. To measure the amount of residency, we want to generate an inventory of how much data of every type is left on the heap, as well as – ideally – why it is being kept there. Fortunately, the GHC runtime already contains an adequate mechanism for generating heap residency profiles, courtesy of Sansom and Peyton Jones [1995]. We can simply adapt their implementation to yield us a statistic of which type of heap object consumed how much memory in total. As noted back in Section 3.5.9 on page 52, heap object info pointers double as code pointers, therefore we can directly treat them as input to our profiling process. Note that looking at the code pointer not only allows us to identify constructors, but also the concrete code point for thunks. This can be especially useful for identifying situations where thunk evaluation gets delayed excessively.

5.8 Analysis

We have now learnt everything there is to know about how to obtain profiling data from Haskell programs using our approach. With our implementation, we can now run arbitrary programs with suitable compilation and runtime parameters and have it produce an eventlog containing both detailed debug records as well as raw sampling data to match. Conceptually we just need to put these two together in order to obtain a full performance analysis.

Yet there are still engineering challenges to consider. Firstly, when we generated the eventlog we put emphasis on making sample generation as light-weight as possible. This almost automatically means that analysis will have to make up for it by processing quite a bit of raw data in order to locate relevant trends. This task is not made easier by the fact that as shown in Section 5.3 on page 121 it is not easy to explain what the data means in the first place. Our aim must be to break the data down in terms of notions that the programmer understands, and work with them in unravelling the causal processes within the program.

5.8.1 ThreadScope

When we explained the requirements for our profiling solution back in Section 2.1.2 on page 7 we defined our role as supporting the user's abductive reasoning process. By nature this process will be incremental, requiring step-wise theory development and evaluation. In order to seamlessly assist the user with this, we therefore want to provide an interactive graphical user interface.

It is therefore fortunate for us that when Jones et al. [2009] implemented the event-log system for GHC, they also developed a sophisticated analysis application, namely ThreadScope. Implementing our profiling analysis as an extension to this tool allows us to make use of a significant amount of existing infrastructure: The application already allows the user to browse an activity profile of the program run, complete with garbage collection statistics. As we will see, the user can allow this to easily browse our performance data.

5.8.2 Debug Maps

However, let us cover some technical details first. The fact that we will have to process quite a bit of data within the analysis tool means that we cannot spend too much time on individual look-ups. This is in fact vital for establishing adequate responsiveness: After all, the sheer volume of the static debug information can easily go up to several hundred megabytes of data for larger applications!

Consequently, the first task for our profiling extension to ThreadScope will be to build indexes for the debug records. Specifically, we construct the following “maps":

Debug Map: All records organised as a tree as defined in Section 5.6.6. This means resolving all IDs used in the event-log, recovering the tree structure.

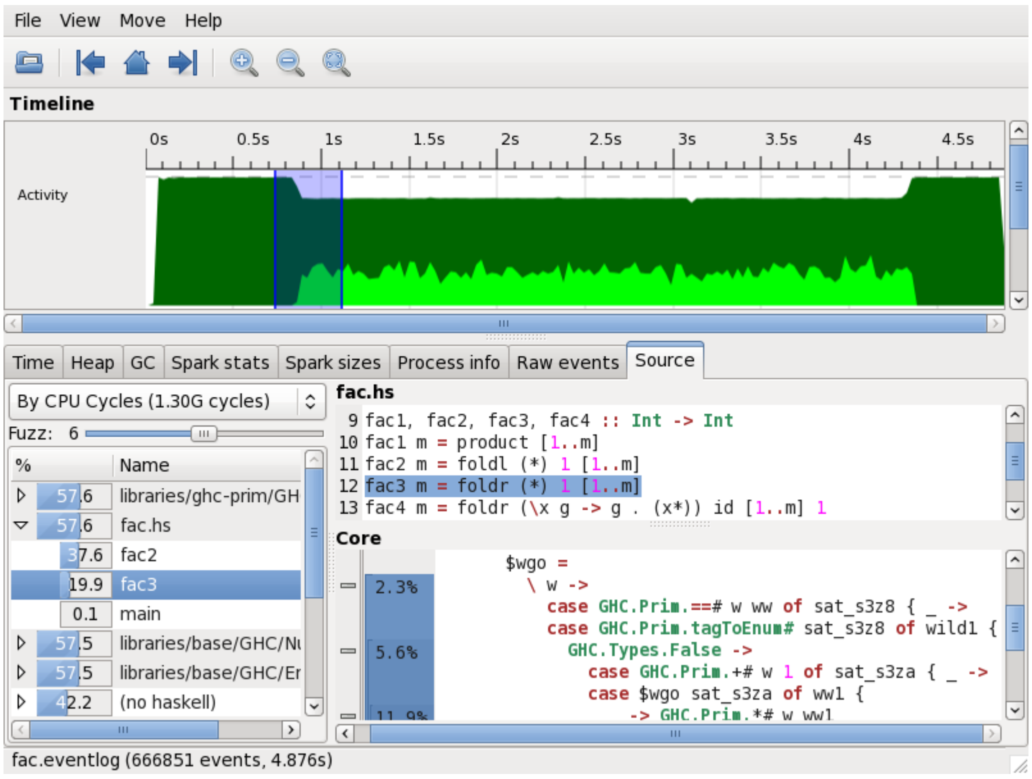
Range Map: A mapping of memory locations to their Cmm blocks. Note that we might end up with memory locations being covered by multiple ranges [16], in which case we chose the most specific one.

Core Map: As explained in Section 5.6.7, we allow the user to save Core code in order to provide a look “inside” the optimised program. This provides fast access to all Core pieces involved.

Sample Map: An index providing quick access to sample messages within the event-log. This will allow us to easily navigate our profiling data, as we can look-up all samples of a certain type in a given time period.

When fully evaluated, these maps index quite a considerable amount of data. In fact, we have been struggling quite a bit with memory overflows simply due to the fact that the maps depend on the complete static debug data block, which is already quite substantial before unpacking. Unfortunately, the architecture of the event log reading library `ghc−events` does not support navigating on-disk data, which reduces our options considerably. We now simply leave data in `ByteString` form for as long as possible, but this is not a viable long-term solution.

[16] This happens if our data was generated by the LLVM backend, and got inlined.



By CPU Cycles (1.30G cycles)

Fuzz: 6

%	Name
57.6	libraries/ghc-prim/GH
57.6	fac.hs
37.6	fac2
19.9	fac3
0.1	main
57.5	libraries/base/GHC/Nu
57.5	libraries/base/GHC/Er
42.2	(no haskell)

```
fac1, fac2, fac3, fac4 :: Int -> Int
fac1 m = product [1..m]
fac2 m = foldl (*) 1 [1..m]
fac3 m = foldr (*) 1 [1..m]
fac4 m = foldr (\x g -> g . (x*)) id [1..m] 1
```

```
        $wgo =
2.3%      \ w ->
            case GHC.Prim.==# w ww of sat_s3z8 { _ ->
            case GHC.Prim.tagToEnum# sat_s3z8 of wild1 {
5.6%          GHC.Types.False ->
                case GHC.Prim.+# w 1 of sat_s3za { _ ->
                case $wgo sat_s3za of ww1 {
11.9%             -> GHC.Prim.*# w ww1
```

fac.eventlog (666851 events, 4.876s)

Figure 5.7: ThreadScope User Interface

5.8.3 Interface Concept

With all profiling data readily accessible, we can face our main challenge for this section: Presenting the information in a way that assists the programmer in solving the performance problem. As we explained in Section 2.1.2 on page 7, for our purposes we think of this process as divided into two stages. First, the programmer will want to gain an overview of the program's performance characteristics. This is both about surveying the collected data as well as comparing it against the expectations. After all, this is where we expect the programmer to come up with a lead: Something that does not fit into the picture, and warrants further investigation. Once a starting point is found, focused data exploration might yield an explanation for the measured phenomenon.

This philosophy is directly reflected in the layout of our user interface. In Figure 5.7 we see a typical view of the ThreadScope profiling tool. Analysis naturally starts at the top "timeline" overview, visualising the whole program run. The programmer will then proceed to hunt for more specific leads in the performance data overview in the bottom-left part of the interface. Once we have an idea what we are looking for, the combined Haskell & Core view on the right bottom side allows drilling deeper into the causal processes at work, ideally yielding an explanation for the problem at hand.

By default, all these information panes will be visible at the same time, which allows the user to jump between them quickly. However, once we get to deeper investigation, the programmer might want to resize and collapse certain parts of the interface. For example, we would typically remove the timeline from view once we have settled for a program phase to investigate – and good knowledge of the source code might make the source view redundant.

5.8.4 Timeline

The time-line view is a standard component of the profiling facilities developed by Jones et al. [2009]. It visualises the program's activity profile over its runtime. Consequently, the data shown is actually not a result of our instrumentation, but is simply won from runtime system status messages emitted into the event log. And even though this data does not say too much about the underlying program, this visualisation can already yield interesting observation about the program. For example, we can often immediately spot certain "phases" in program evaluation, such as pronounced "set-up" periods for loading data into memory. Furthermore, ThreadScope introduces information about time spent on garbage collections into the graph, which an experienced user can also use to assess the general heap allocation behaviour of the program.

Apart from providing the "big picture" view of our data, the timeline is also our first interactive user interface component. Simply by selecting a portion of the profile, the user is able to focus the investigation on a certain part of the complete profile. The tool also actively helps to identify program phases: For example, after selecting fac3 in Figure 5.7 the program added bright green markings to the program's activity profile to highlight the phase where this function was most active. The reason such a phase exists is because the program benchmarks a number of factorial implementations:

```
fac1, fac2, fac3, fac4 :: Int → Int
fac1 m = product [1..m]
fac2 m = foldl (*) 1 [1..m]
fac3 m = foldr (*) 1 [1..m]
fac4 m = foldr (λx g → g . (x*)) id [1..m] 1
main = forM_ [fac1,fac2,fac3,fac4] $ λfac →
        replicateM_ 100 $ forM_ [4000..5000] $ λn →
         evaluate (fac n)
```

Listing 5.47: Factorial Implementations

So it is no surprise that fac3 is only active during a part of the program run. However note that our tools will be able to identify and investigate such phases even where they arise indirectly from the program's behavior.

5.8.5 Performance Data

However, for present purposes we focus on how to present our own performance data. For this, we first need the user to select which kind of samples they would like to view. After all, as we saw in Section 5.7 on page 162 we might have a significant number of different sampling methods at our disposal. Just like with the time-line, this allows us to shift the focus to different aspects of the program's performance characteristics. For example, in Figure 5.7 we see that "by CPU cycles" is selected, which means that we are viewing data sampled by the CPU cycle counter, via the `perf_events` profiling back-end. Using such general-purpose verbs is generally a good idea to get a first impression of the "hot spots" within the program.

Once the data set of interest is selected, our profiling tool can make use of the prepared debug maps: The sample map will tell us where we find the samples in question, and doing a range map lookup on the sample values gives us a total "weight" for each individual back-end block – a proper low-level program profile. However for the purpose of presenting an overview this is far too low-level. After all, the structure of back-end blocks will at best correspond loosely to the program that was originally written. Consequently, this is the point where we want to use source code links to better aggregate our data.

We explained our general approach to this problem back in Section 5.3.4 on page 125: Given a debug block we can reconstruct a number of causal relationships to source code locations simply by making use of the static context of the block. After all, we took great care in Section 5.6.6 on page 156 to organise blocks in a way that we retain information about which set of blocks every tick scopes over. However, this still leaves us with two problems. Firstly, due to our approach we will inevitably end up in situations where we have more than one source tick associated with a given performance figure. At this point we have basically the choice of either heuristically selecting a most "significant" one – as we did in Section 5.6.4 on page 152 – or simply represent the data as we see it.

As it is our philosophy that performance is often a product of multiple disconnected pieces of code interacting, we choose the latter approach. As a result, we can see in Figure 5.7 that multiple list entries end up with more than 50 % of the cost allocated to them. We can especially see that the package `ghc−prim` ends up with a rather large amount of cost. And this is hardly surprising, given that we expect most code to causally depend on how primitive operations are defined. An extreme example here is the `($)` Haskell operator, which due to its use in program layout is so ubiquitous that it is not uncommon to find it associated with a large chunk of performance data. The user will have to take care to disregard such outliers, as from the viewpoint of our profiling tool it is a perfectly valid concern that the wide-spread use of `($)` might some day cause a subtle and well-spread-out performance problem.

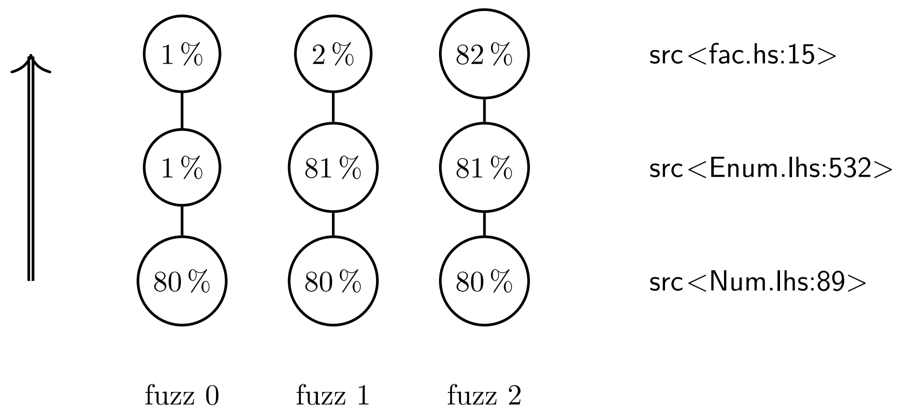


Figure 5.8: Performance Data Fuzz

However, even given that we are willing to associate multiple source code links with a given sample, we still have a choice in how far we want to go. After all, as we argued in Section 5.3.1 depending on context some causes can end up being more telling about the underlying events than others. Specifically, we know that the closer we find a source code annotation to our block, the more likely it is that it will be specific enough to identify it in the user's eyes. This is a rather two-edged sword. After all, by this logic the most specific source code link might well be the definition of, say, a multiplication operation. For example consider Figure 5.8, where each node corresponds to a block of code, annotated with the amount of cost we associate with it. For the scope path on the left, the "hot spot" is clearly in the bottom-most "leaf" block, which however only carries a source code annotation to Num.lhs. In order to establish a connection between this performance data and the original source code in `fac.hs`, we therefore have to "fuzz" the performance data by propagating it towards the root of the block scope tree. Note though that weakening the narrowness of the mapping also substantially increases the spread of our performance data: Now the source code from Enum.lhs also gets attributed with more than 80 % of the cost in question. Choosing the right "sweet" spot here is again a task that we will have to leave to the user [17]. In our example we chose a fuzz of 6, which is exactly enough to be able to track all cost to the originating top-level function.

Finally, due to our inclusiveness the total list of source code locations associated with performance data can become quite long and hard to manage. We remedy this issue by structuring the source code locations by source module, associating another aggregated performance statistic with their respective entry. For example, we see in

[17]While our interface choice might be somewhat unconventional, the act of subsuming performance data upwards in "stacks" is something that most profilers support. Keeping in mind that we are not actually using call stacks, we can loosely liken a fuzz 0 to "individual cost", and fuzz ∞ to "inherited cost" for cost-centre profiling.

Figure 5.7 that 97.3 % performance data could be mapped to some source tick from the `fac.hs` module. Structuring the data like this is quite useful, as the spread of the performance data across modules already tells us a rough story about what the program is currently doing. Furthermore once we put our focus on a certain source file, the structured list allows us to quickly access related source code nouns.

5.8.6 Source View

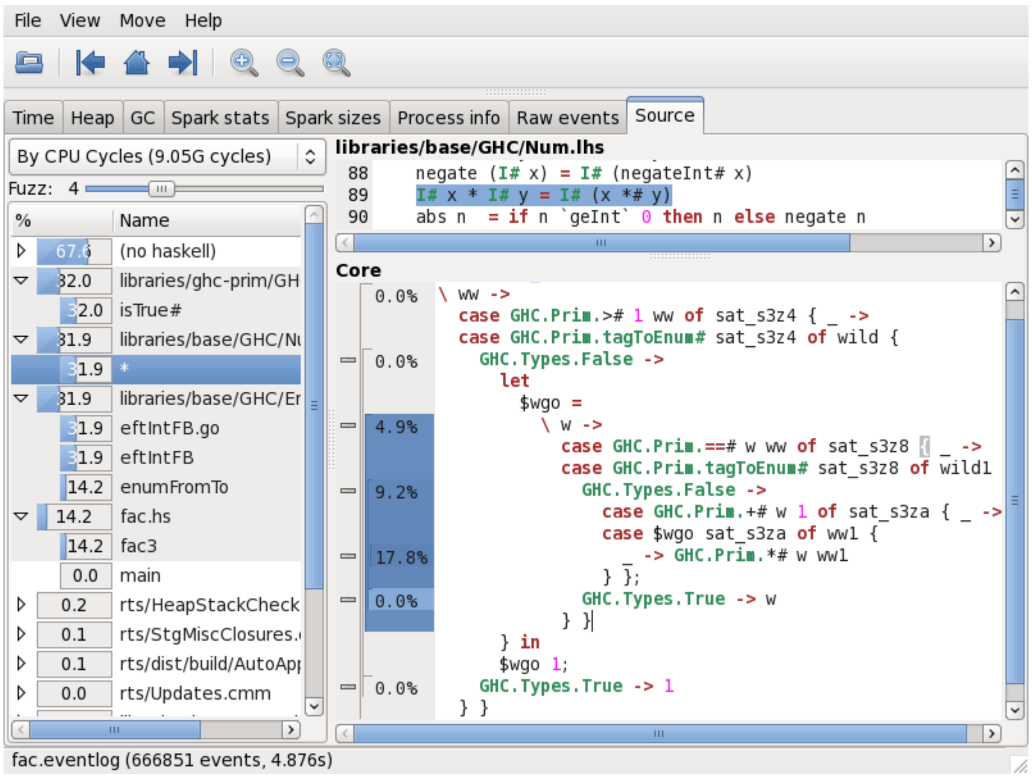
Up to this point, we have simply used the name attribute of source notes to refer to source code locations. As defined in Section 5.4.2, this name is composed from program names and likely to allow the programmer to quickly identify the source of the referenced code. However, once we have a concrete performance problem that we want to investigate, we will not only be interested in the identity of the involved functions, but also in their concrete implementation. In order to reduce the amount of footwork involved, we integrate this directly into our user interface: When a source noun gets selected we automatically look for the associated source file, and highlight the appropriate source code portion if successful. This does not always have to be the whole function – depending on the current fuzz value and the size of the static context, we might only highlight a part of the function. This could for example be a parameter that got inlined into an inner loop, such as the operation passed to a `map` function.

5.8.7 Core View

Using a source-based profiling overview works well while we are trying to get an overview of our performance data. However in subsuming the costs by source annotation we are actually throwing away information. After all, simple source code links will tell us that resource usage could be linked to a combination of source nouns – but not how and in which context these source nouns interacted. When calculating the factorial function, we would hardly be surprised by the finding that most cost is associated with the factorial function, number enumeration or multiplication operations. Relating performance statistics to bare source code locations does not *explain* the problem very well.

Fortunately, our sampling back-ends collects data at a significantly finer granularity: Pointers into the object code allow us to theoretically pin-point resource usage at the accuracy of a single instruction. In our case we bound debug-information on the block level in Section 5.6 on page 149, which reduces the precision somewhat. On the other hand the implementation of a single Haskell function can easily consist of hundreds of back-end blocks, so this is still quite a step up in accuracy.

This leaves the question of how we should present the data. As we introduced back in Section 2.2.3 on page 11, the main way that we can assist the programmer at this



By CPU Cycles (9.05G cycles)

%	Name
67.6	(no haskell)
32.0	libraries/ghc-prim/GH
32.0	isTrue#
31.9	libraries/base/GHC/Nu
31.9	*
31.9	libraries/base/GHC/En
31.9	eftIntFB.go
31.9	eftIntFB
14.2	enumFromTo
14.2	fac.hs
14.2	fac3
0.0	main
0.2	rts/HeapStackCheck
0.1	rts/StgMiscClosures.
0.1	rts/dist/build/AutoApp
0.0	rts/Updates.cmm

fac.eventlog (666851 events, 4.876s)

libraries/base/GHC/Num.lhs

```
    negate (I# x) = I# (negateInt# x)
    I# x * I# y = I# (x *# y)
    abs n  = if n `geInt` 0 then n else negate n
```

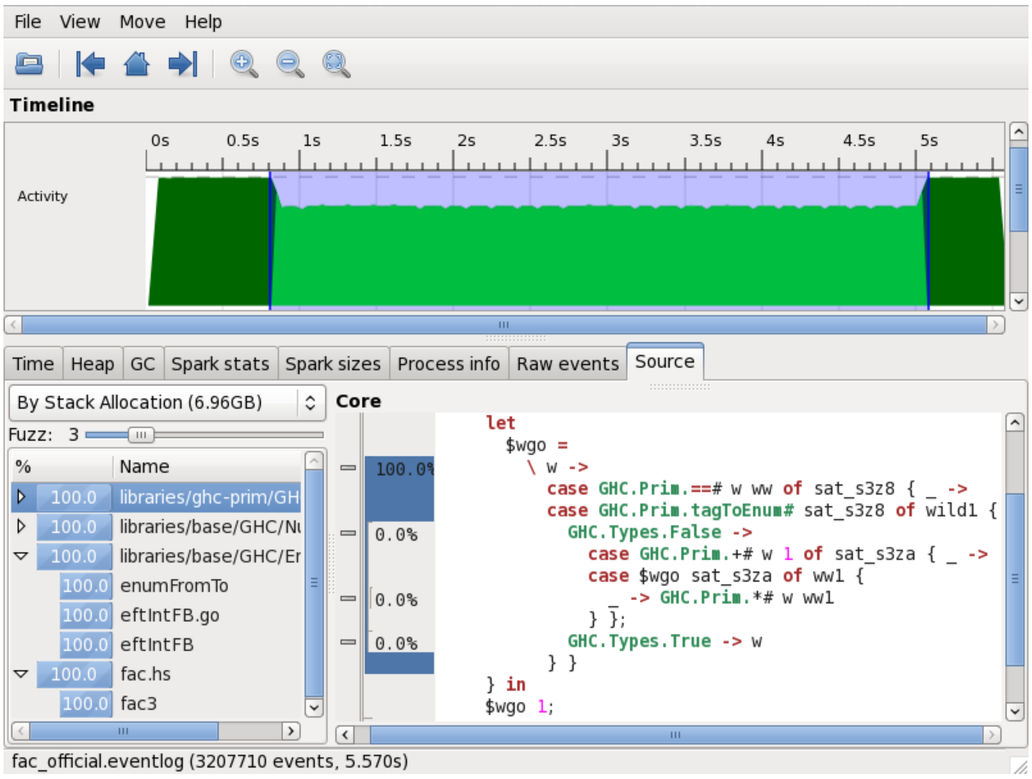
Core

```
 0.0%   \ ww ->
          case GHC.Prim.>># 1 ww of sat_s3z4 { _ ->
          case GHC.Prim.tagToEnum# sat_s3z4 of wild {
 0.0%       GHC.Types.False ->
              let
                $wgo =
 4.9%             \ w ->
                    case GHC.Prim.==# w ww of sat_s3z8 { _ ->
                    case GHC.Prim.tagToEnum# sat_s3z8 of wild1
 9.2%                 GHC.Types.False ->
                        case GHC.Prim.+# w 1 of sat_s3za { _ ->
                        case $wgo sat_s3za of ww1 {
17.8%                     _ -> GHC.Prim.*# w ww1
                        } };
 0.0%                 GHC.Types.True -> w
                    } }
              } in
              $wgo 1;
 0.0%       GHC.Types.True -> 1
        } }
```

Figure 5.9: Core View

point is by making the compilation process more transparent. This would be basically impossible to convey just by pointing out certain source code locations. Instead, we have to speak the same language as the processes that make these decisions, which in this case means the Core intermediate language. This choice has a number of advantages: Not only is it a good middle ground between the high-level Haskell source code and the thoroughly obfuscated back-end code, but as we showed back in Section 3.5 on page 41 it is also not too hard to predict performance characteristics at this level.

Reconstructing the Core code means that we need to reassemble and pretty-print the CoreNote fragments we generated in Section 5.6.7 on page 158. Figure 5.9 shows what this would look like for our running example: On the left-hand side we annotate how much resource usage we found at every visible location. For our example, we see that we spent 4.9 % entering the worker function $wgo and comparing the counter w against the upper limit ww. However most of the cost happens once we have determined that we need to recurse: Incrementing w and performing the recursive call consumes 9.2 % of our time, while we spend 17.8 % of our time simply performing multiplications on the return value. As we observe on the left, the remaining 67.6 % processor time are actually spent inside the runtime system, which struggles to remain efficient given the amount of strain we put on the stack allocator.


File View Move Help
Timeline
Activity
0s 0.5s 1s 1.5s 2s 2.5s 3s 3.5s 4s 4.5s 5s
Time Heap GC Spark stats Spark sizes Process info Raw events Source
By Stack Allocation (6.96GB)
Fuzz: 3
% Name
Core

%	Name
100.0	libraries/ghc-prim/GH
100.0	libraries/base/GHC/Nu
100.0	libraries/base/GHC/Er
100.0	enumFromTo
100.0	eftIntFB.go
100.0	eftIntFB
100.0	fac.hs
100.0	fac3

```
100.0%      let
              $wgo =
                \ w ->
                  case GHC.Prim.==# w ww of sat_s3z8 { _ ->
                  case GHC.Prim.tagToEnum# sat_s3z8 of wild1 {
0.0%                GHC.Types.False ->
                      case GHC.Prim.+# w 1 of sat_s3za { _ ->
                      case $wgo sat_s3za of ww1 {
0.0%                    _ -> GHC.Prim.*# w ww1
                      } };
0.0%                GHC.Types.True -> w
                  } }
            } in
            $wgo 1;
```

fac_official.eventlog (3207710 events, 5.570s)

Figure 5.10: Tracking Stack Allocation

This means that we have actually come very close to identifying the source of inefficiency in the function: Having this much cost on the “return path” is already suspicious. However, we can do even better: If we go back and use the “stack allocation” verb instead, we see in Figure 5.10 that the picture becomes even more clear. The stack allocation shown is 6.96 GB, which equals 1.2 GB/s or about 69 kB per evaluation of fac3 [18]! Furthermore, we can actually see that virtually all of the stack allocation originates from the function entry point of the worker function $wgo.

5.8.8 Core Tools

For working with Core we have to keep in mind that it is still primarily an intermediate language of the Haskell compiler. This means that readability is not one of its design goals, and we can not count on Core code being as compact as it was in the last section. In fact, a number of Haskell optimisations such as in-lining, worker/wrapper [Gill and Hutton, 2009] and call pattern specialisation [Peyton Jones, 2007] have a tendency to blow up the code size up significantly. Especially the latter optimisations tend to

[18] About what we would expect from Listing 5.47 on page 171, as this is just a bit less than the 72 kB required for 4500 stack frames of 16 bytes each. The remaining difference comes from the fact that the program will never deallocate its initial stack chunk.

produce Core code where we have a lot of set-up code surrounding a relatively small inner loop. While this is a great idea for performance, navigating such Core trees can feel like being stuck in a maze.

Fortunately, we can again count on the Pareto principle, which as introduced in Section 2.1.2 on page 7 states that most of the time, our performance problem will be located in a rather small portion of the complete program. This applies to Core code too: While the offending piece of code might be buried deep within the Core tree of a big function, it is quite likely that we will also find resource consumption in close proximity. For the purpose of our profiling tool, it is therefore essential that we allow the user to easily navigate around the tree. In order to avoid distraction, we should especially never display too much information by default.

Fortunately, tree navigation is a well-studied field in information visualisation [Herman et al., 2000]. In fact, we can cut out most information simply by

1. Limiting ourselves to one concrete sub-tree at a time and
2. "folding" child sub-trees that are not of interest.

By default, the sub-tree we choose to show is the one that contains all sampling data over a certain threshold [19]. As a result, if an "inner loop" is found deep within a wrapper function, the user never needs to see these wrappers in the first place. Along the same lines, we automatically fold away any piece of code that does not contribute to the currently considered profile portion. This means that any function or `case` alternative that seemingly has no connection to our performance problem gets removed from view automatically. However note that these are only default choices. If the need arises, the user can override each of these decisions: The left "gutter" of the Core view allows the user to both extend the view by moving the root of the shown tree "upwards" as well as folding and un-folding arbitrary parts of the code.

Finally, it has proved to be useful if we allow the user to query the types of variables within the Core code. This can be used not only for identifying rare type-based performance problems as shown in Section 3.3.2 on page 28, but also for making sense of variables within the Core code. After all, it is not uncommon for program transformations to significantly garble variable names, yet their type often stays predictable. In order to make our Core view as compact as possible we do not show these types by default, but instead make them available on request as a tool-tip.

[19] Actually, we ascend until the next function definition. This way we get a better feel for the function's overall control flow, as well as keeping parameter names in view.

Chapter 6

Evaluation

"Young man! I make the classical remarks around here, alea iacta est and all that, and what is more, you have not answered my question: What are you going to do with all these menhirs?"

— *Obelix and Co., René Goscinny & Albert Uderzo*

We have covered the theory behind our profiling solution and have explained how we have brought it to life. As far as our design work is concerned, this should have yielded us a tool that is useful in a wide variety of situations. After all, we took care to track causality without regard for the nature of the connection. Only once we hit technical limits we allowed ourselves to compromise on this principle. Furthermore, we identified a large set of verbs, corresponding to an equally rich family of potential performance problems that we might want to track down.

However, it is quite clear the we can not meaningfully claim completeness: In practice constructing full causality terms is simply unrealistic, and we will not be able to collect samples for every interesting verb. The reason is that we have limits on our own resource consumption: Virtually everything we do comes with a price tag. Tracking causes at compile time will slow down compilation and increase the size of produced binaries. Collecting a useful amount of sampling data might slow down the program, and introduce skews in the process. And in the end, we might even find out that one of our assumptions turned out to be wrong, rendering us unable to break the performance problem down with verbs or connect it the original cause nouns.

While we cannot prove usefulness, we can still attempt to ease our doubts by showing that our tool does the right thing in realistic scenarios. Section 6.1 will start with looking at the performance of our profiling tool itself. Our goal will be to show that we can produce the desired debug records and samples in a realistic setting. In Section 6.2 on page 190 we proceed to walk through a usage scenario of our profiling tool for a small real-world performance problem.

6.1 Performance

It might seem redundant to stress that we have to care about the performance of our profiling solution. After all, the quicker we can be about our results, the faster the user will be able to arrive at a solution. However, we are actually under especially high pressure to deliver good performance. After all, instrumentation is a secondary concern for the program. Furthermore, as we argued back in Section 2.1.2 on page 7 the actual performance optimisation process often consists of a lot of trial-and-error on the user's part. This means that we should not only perform well – ideally our work should be virtually invisible to the user.

6.1.1 Test Data

In order to identify the overheads of our profiling framework systematically, we will use the "nofib" benchmark suite [Partain, 1993] as our reference. For our purposes this benchmark suite features a large set of Haskell programs, which are largely regarded as typical representatives for the even greater variety of Haskell code out there. Our general approach will simply be to run this benchmark suite with different compiler and runtime systems options, and observe how our results change as a result.

We will run all benchmarks on a personal computer running CentOS 6.5, with an Intel® Xeon® CPU clocked at 3.1 GHz and 32.1 GB system memory. As virtually every run will generate too much data to show in the context of this document, we will generally only show a few representative results. Averages and standard deviations will be for the complete data set. Where we consider run times, we will however exclude small results ($< 0.2\,\mathrm{s}$) in order to reduce noise.

6.1.2 Compilation Overhead

The first point where our profiling solution comes into play is during the compilation of the program. Recall from Section 5.4 on page 129 forward that we generate various Tickish annotations that will be kept around throughout all code transformation stages, eventually culminating in the generation of debug sections for the final binary. We are willing to accept a moderate slowdown at this point. After all, our annotations affect some of the most busy data structures used by the compiler: intermediate program representations. This means that every time the compiler reconstructs a certain Core expression or Cmm block, we will have to generate copies of Tickish nodes or CmmTick pseudo-instructions, to say nothing about the complexity involved in making sure that optimisations can look through them correctly.

Let us test this using our benchmarks. We will run the four configurations shown in Figure 6.1. Each set of compiler command line options will result in more informa-

	Program	Libraries	Description
Base			Baseline build without annotations
Dbg	`-g`		Only annotations from program source
Libs	`-g`	`-g`	Annotations from program and libraries
Core	`-g -fsave-core`	`-g`	Core gets retained for analysis

Figure 6.1: Benchmark Configurations

		Base	**Dbg**	**Libs**	**Core**
		[s]	[%]	[%]	[%]
cacheprof	Arch_x86	0.95	+15.8	+15.8	+16.8
	Generics	0.28	+21.4	+28.6	+32.1
	Main	3.05	+12.1	+20.0	+21.6
calendar	Main	0.40	+7.5	+17.5	+17.5
cichelli	Auxil	0.24	+12.5	+16.7	+20.8
	Prog	0.22	+13.6	+27.3	+31.8
circsim	Main	1.04	+10.6	+19.2	+22.1
clausify	Main	0.42	+7.1	+14.3	+16.7
comp_lab_zift	Main	0.91	+12.1	+20.9	+24.2
compress	Decode	0.25	+8.0	+24.0	+28.0
	Encode	0.26	+7.7	+11.5	+15.4
compress2	Encode	0.53	+17.0	+20.8	+34.0
	−1 **s.d.**	—	+5.1	+13.5	+16.4
	+1 **s.d.**	—	+17.9	+26.4	+30.6
	Average	—	+11.3	+19.8	+23.3

Figure 6.2: Compilation Time

tion being available for later analysis. For our purposes "Core" is the most complete configuration, having full source code annotations as well as Core notes for deep performance analysis as explained in Section 5.6.7 on page 158. We do not consider compiling libraries with Core notes here, as Core annotations in libraries make no difference for modules using them.

The results are shown in Figure 6.2. The modules listed were chosen arbitrarily, but we made sure to not show modules with low compilation times ($< 0.2\,\mathrm{s}$). As expected, we are facing a moderate slow-down of between roughly 11.3 % for simple source code annotations and 23.3 % for complete Core information. As expected, the performance hit is definitely measurable, but fortunately not too severe to pose a problem for usability. Our price tag might in fact be low enough to warrant keeping debugging on by default for anything but production builds.

			Dbg	**Libs**
		[Terms]	[Ticks]	[Ticks]
cacheprof	Arch_x86	4733	937	1729
	Generics	1105	568	912
	Main	10868	3851	7722
calendar	Main	1087	249	956
cichelli	Auxil	1067	344	567
	Prog	827	239	524
circsim	Main	3563	1212	2511
clausify	Main	1328	398	672
comp_lab_zift	Main	3441	1395	2099
compress	Decode	924	210	460
	Encode	2262	620	709
compress2	Encode	4148	824	1479
	WriteRoutines	804	314	534
constraints	Main	2074	420	1073

Figure 6.3: Tick Count

6.1.3 Tick Counts

However, we do expect to get something in return. After all, the usefulness of our profiling approach depends on a suitable granularity of source code annotations. This begs the question: How high was the tick concentration in the given programs in the first place? Let us attempt to answer this for the Core level. Figure 6.3 shows the number of ticks after Core simplification (`-ddump-simpl`)e for configurations "Dbg" and "Libs". We skip "Base" and "Core" here, as their tick count will be 0 and the same as "Libs" respectively. For comparison, we have put the Core "size" according to CoreStats on the left-hand side, which roughly corresponds to the number of expressions present in the program.

We observe that while the concrete proportion fluctuates somewhat, it stays around 1:2 to 1:5. We can take this as a encouraging sign that programs will tend to likely neither be over- nor under-annotated. Finding stability here is not entirely surprising: As we noted on several occasions, ticks tend to accumulate at characteristics control flow points within the program's Core code. As introduced in Section 5.4.4 on page 132, this will often allow us to merge ticks, leading to a soft self-regulation of tick counts. We can therefore say with some confidence that at the Core level annotations seem to act as expected. Transformations leave us enough source code links to work with in the following stages.

		Base [kB]	Dbg [%]	Libs [%]	Core [%]
cacheprof	Arch_x86	341	+84.4	+96.8	+117.5
	Generics	64	+183.9	+208.2	+261.3
	Main	586	+165.1	+196.0	+247.2
calendar	Main	47	+157.1	+223.2	+281.1
cichelli	Auxil	41	+196.0	+224.1	+297.0
	Prog	45	+165.2	+200.3	+262.3
circsim	Main	146	+181.4	+224.5	+317.0
clausify	Main	58	+224.8	+248.3	+306.2
comp_lab_zift	Main	162	+204.5	+228.3	+293.2
compress	Decode	51	+164.9	+187.2	+259.5
	Encode	77	+55.3	+60.7	+83.7
compress2	Encode	142	+204.5	+225.6	+335.3
	WriteRoutines	29	+223.8	+263.6	+353.9
constraints	Main	85	+185.0	+222.1	+294.6
	−1 **s.d.**	—	+105.9	+120.5	+150.3
	+1 **s.d.**	—	+215.0	+263.1	+357.2
	Average	—	+154.6	+183.0	+238.3

Figure 6.4: Module Object File Sizes

6.1.4 Binary Size

During compilation, the annotations will get transformed into the various debugging information representations explained back in Section 5.6 on page 149. As our goal is to affect program execution as little as possible by default, this information will be completely contained in separate object file sections. This has the interesting side effect that if we now use standard tools like `strip` or `objcopy` to strip these sections away, we actually obtain essentially the same program we would have arrived at if we had not compiled with debug annotations in the first place! This level of separation makes it quite plausible that the increased size itself will only have an negligible effect on program performance.

However even given these considerations, program size is far from unimportant. Building programs with GHC is already notorious for resulting in rather large executables, due to statically linking all libraries by default. What we are doing is likely to make this situation even worse. And as we see in Figure 6.4, the effect is quite pronounced: Compilation with debug annotations will blow up the size of produced object files by a factor of two to four! Even with storage space being plentiful in this day and age, this is quite a pill to swallow.

To investigate where the blow-up is coming from, let us look at the sections of a module compiled with the "Core" configuration. Using `objdump -h` on the "Encode" module from "compress2" yields us essentially the following size distribution:

```
Name           Size
.text          0001284f
.data          00000470
.rodata        0000010f
.debug_info    0001205e
.debug_abbrev  0000002c
.debug_line    00001adf
.debug_frame   00003850
.debug_ghc     0003611f
```

Listing 6.1: Object File Sections

Clearly two debug sections contain the lion's share of the data: `.debug_info` and `.debug_ghc`. Consider debug information first: As explained back in Section 5.6.2 on page 150, this section contains a record for every single compilation unit, procedure and block in the object file. This does not make it hard to believe that it might quickly grow to a remarkable size: After all, the GHC native code generation will often generate rather small blocks, so it is completely plausible that we might end up generating more debug information into `.debug_info` than actual machine code into the `.text` section. This is even worse for the `.debug_ghc` section: Not only will this contain information about every single block, but also potentially source code links as well as the original Core code. It is therefore not surprising that even with a somewhat compact encoding we end up taking quite a significant chunk of space.

In the end, we have to admit that in order to perform the kind of performance analysis that we have in mind, we have to pay a certain prize. Our best argument at this point is that all this cost is optional and can be stripped at any point, letting the user decide how much space concerns matter to her.

6.1.5 Core Size

Apart from performance concerns, this is actually a point where we can check back with one of our design goals. After all, when we explained in Section 5.4 on page 129 how we will deal will GHC's transformations, we did so with the intention to be exhaustive: Every single optimisation present in GHC should still fire with debug annotations present. This is the property that would truly allow us to claim that we have a chance of spotting every performance problem out there – even if it depends on certain optimisations running.

		[Terms]	**Dbg** [%]
cacheprof	Arch_x86	4733	0.0
	Generics	1105	+2.9
	Main	10868	+0.1
calendar	Main	1087	0.0
cichelli	Auxil	1067	0.0
	Prog	827	0.0
circsim	Main	3563	0.0
clausify	Main	1328	0.0
comp_lab_zift	Main	3441	-2.4
compress	Decode	924	0.0
	Encode	2262	0.0
compress2	Encode	4148	0.0
	WriteRoutines	804	0.0
constraints	Main	2074	-1.3
	-1 **s.d.**	—	-1.8
	$+1$ **s.d.**	—	+1.7
	Average	—	-0.1

Figure 6.5: Core Size

However, as we see in Figure 6.5, our work up to this point is not quite enough: While the Core size stays the same for most modules, we can see a few instances where the Core code seems to change. This demonstrates how hard it is to claim completeness for a compiler as complex as GHC. For example, here is one of the issues:

```
f = λx y → (,) x y
```

Listing 6.2: Constructor Application

where (,) is the constructor of the pair. Here we could η-reduce f by setting f = (,). This is a really minor optimisation, as the only difference here is that slightly less object code gets generated. Apart from this, there are further issues, for example with fully supporting the call pattern specialisation [Peyton Jones, 2007].

6.1.6 Run Time Overheads

The reason that we care so much about optimisations in the first place is of course because they influence runtime performance of our program. While good behaviour at compile time is a nice bonus, what happens at runtime will make or break the usefulness of our whole profiling infrastructure. No matter whether it is due to missing optimisations or sampling overheads – if it ends up altering the performance of the

	Compilation	**Runtime**	**Description**
Base			Baseline without eventlog generation
Dbg	`-g`		Simple build with debug annotations
Ev	`-eventlog`	`-ls`	Eventlog gets generated
DbgEv	`-g -eventlog`	`-ls`	Eventlog with debug records

Figure 6.6: Runtime Configurations

	Base [MB]	**Dbg** [%]	**Ev** [%]	**EvDbg** [%]
k-nucleotide	3921	0.0	0.0	0.0
integer	3338	0.0	0.0	0.0
pidigits	2885	0.0	0.0	0.0
constraints	2104	0.0	0.0	0.0
cryptarithm1	2051	0.0	0.0	0.0
n-body	1600	0.0	0.0	0.0
circsim	1299	0.0	0.0	0.0
hidden	1084	0.0	0.0	0.0
binary-trees	956	0.0	0.0	0.0
scs	917	0.0	0.0	0.0
fannkuch-redux	870	0.0	0.0	0.0
transform	680	0.0	0.0	0.0
exp3_8	597	0.0	0.0	0.0
atom	537	0.0	0.0	0.0
−1 **s.d.**	—	-0.0	-0.0	-0.0
+1 **s.d.**	—	+0.0	+0.0	+0.0
Average	—	-0.0	-0.0	-0.0

Figure 6.7: Allocation Overhead

program too much, all of our collected data might turn out to be nothing but smoke and mirrors. This is why it is especially important that we take a look at the runtime performance of the compiled programs. Even if we cannot guarantee the Core to be completely identical, a good performance match across the benchmark suite would be an encouraging sign that we are not likely to encounter any significant “heisenbugs” that vanish upon inspection. This goes not only for our own debug annotations, but also for the existing eventlog infrastructure, which we are going to use for generating program profiles. The four configurations in Figure 6.6 cover different combinations of these debug options.

The results are shown in Figure 6.7 and Figure 6.8. The first performance metric shown is allocation amount. As the allocation amount of a given Haskell program is mostly deterministic, this is an excellent metric for comparing two Haskell program

	Base	**Dbg**	**Ev**	**DbgEv**
	[s]	[%]	[%]	[%]
k-nucleotide	5.0000	+0.5	-0.1	+0.4
fannkuch-redux	3.6467	-0.2	-0.1	0.0
spectral-norm	2.4550	0.0	-0.2	-0.2
integer	1.5300	-0.8	+0.7	-2.6
n-body	1.1300	0.0	-0.4	-0.9
constraints	0.6600	+2.0	+0.3	+1.5
cryptarithm1	0.4350	+1.1	-1.1	-1.1
fasta	0.3850	-0.4	-1.3	-1.3
pidigits	0.3700	0.0	-1.8	-2.7
circsim	0.3633	+1.8	-0.9	-0.5
binary-trees	0.3600	+2.8	+6.0	+8.3
hidden	0.3000	0.0	+3.3	+1.1
wheel-sieve1	0.2783	+0.6	-3.0	-3.0
scs	0.2517	+3.3	-0.7	+3.3
−1 **s.d.**	——	-1.8	-3.9	-4.1
+1 **s.d.**	——	+2.0	+2.0	+2.5
Average	——	+0.1	-1.0	-0.9

Figure 6.8: Mutator Time Base Overhead

	RTS options		RTS options
Base	`-ls`	**Cycle**	`-ls -Ey`
Time	`-ls -Et`	**Cycle2**	`-ls -Ey100000`
Time2	`-ls -Et100`	**Cache**	`-ls -Ec`
Alloc	`-ls -Ea`	**Cmiss**	`-ls -EC`
Stack	`-ls -Es -kc8k`	**Branch**	`-ls -Eb`
Heap	`-ls -Eh`	**Bmiss**	`-ls -EB`

Figure 6.9: Sampling Configurations

runs for functional differences. And as we can see, for this statistic we manage a pretty much perfect match between all configurations.

However, things get less predictable when we consider actual run time. Note that in Figure 6.8 we deliberately consider only "mutator" time, which is the program time excluding start-up time and garbage collections. This is a good idea here, as copying debug records into the eventlog might cause a significant amount of delay when starting the program. Furthermore, we are not going to profile garbage collections, therefore possible performance changes there do not concern us. Even without these factors, it is quite clear that the results are still quite noisy: Paradoxically, on average performance even improves with event-logging active. However, the good averages should show that we have not introduced any severe biases, and can sensibly profile the code resulting from either configuration.

6.1.7 Sampling Overhead

The next important influence on program performance that we have to acknowledge is sampling overhead: The act of capturing and emitting samples causes performance to be lost, therefore any significant overhead could introduce skews. Figure 6.9 shows an overview of the sample collection configurations that we will test. Due to our work in Chapter 5 we have quite a few back-ends to choose from, with `perf_events` alone providing 6 different hardware performance counter configurations (on the right). On the other hand, the left hand side represents the 3 remaining sampling approaches, using timers, allocation and heap residency profiling respectively. Also note that we will adjust the sampling rate in a few cases: For the configurations "Time2" and "Cycle2" we overrode the sampling rates to 100 μs and 100 000 cycles, which are 10 times higher than their respective defaults. Furthermore, for stack allocation we decrease the stack block size (8 kB, default 32 kB) to force more stack overflows and therefore samples.

In Figure 6.10 and Figure 6.11 we have collected data about the sampling overhead of using the different sampling back-ends. Because we are generally interested in the running Haskell code, we again focus on mutator time. The exception is heap profiling, which actually runs as part of the garbage collection pass and therefore does not influence mutator time. Consequently we switch to comparing runtime overhead. The results work out as expected: Moderate slowdowns across the board. The exact amount varies quite a bit due to run-time noise, but we generally stay well below 5 % overhead. The most notable outlier is the "Cycle2" configuration, where the low sampling rate caused costs to spike up noticeably. On the other end of the spectrum we have the "Alloc" configuration, which is cheap enough to incur basically no overhead. As the "Stack" configuration uses the same mechanism, we would normally expect the same good performance characteristics. However the increase in stack block size can have quite dramatic effects on program performance, as the +32.1 % outlier for "ansi" should demonstrate.

6.1.8 Average Overhead

These performance measurements should convince us that all of our profiling back-ends have the ability to work well alongside program evaluation. However, we still have to show that each back-end was successful in collecting enough sample data in order to inform performance analysis. Figure 6.12 therefore shows us two statistics about the generated data: First we give the average sampling rate over all program runs (weighted by run time). Heap profiling achieves a very high rate as this point, as it will set weights based on heap object size – which basically makes every single byte on the heap a sample. The next notable configuration is, again, heap allocation profiling, which

	Base	**Time**	**Time2**	**Alloc**	**Stack**	**Heap***
	[s]	[%]	[%]	[%]	[%]	[%]
k-nucleotide	49.76	-0.3	+1.5	-0.2	-0.5	-0.7
fannkuch-redux	48.67	+0.5	+2.0	+0.0	+0.0	+0.0
n-body	11.23	+0.6	+2.3	+1.0	-0.0	+0.8
binary-trees	8.42	+1.3	+3.7	+0.2	+0.4	+76.7
spectral-norm	8.23	+0.1	+1.5	-0.1	-0.1	-0.1
fasta	3.79	+0.5	+5.5	+0.3	0.0	+0.0
integer	1.49	+0.2	+2.7	0.0	0.0	+0.3
exp3_8	1.26	0.0	+2.5	+0.3	+7.5	0.0
pidigits	1.06	0.0	+3.8	-0.3	-0.2	+0.0
wheel-sieve1	0.92	0.0	+2.2	0.0	+0.5	+7.4
reverse-complem	0.83	0.0	+2.2	0.0	-0.4	-0.4
kahan	0.81	-0.4	+1.6	0.0	-0.4	+0.0
knights	0.69	+1.4	+2.9	0.0	0.0	+1.5
ansi	0.56	+0.6	+3.5	+1.5	+32.1	+0.5
primes	0.48	0.0	+2.1	0.0	+10.4	+0.2
-1 s.d.	—	-0.5	+1.9	-0.4	-3.9	-5.0
+1 s.d.	—	+1.6	+4.7	+1.2	+9.5	+12.0
Average	—	+0.6	+3.3	+0.4	+2.6	+3.1

(* Heap configuration compares run time)

Figure 6.10: Sampling Mutator Time Overhead (1/2)

	Base	**Cycle**	**Cycle2**	**Cache**	**Cmiss**	**Branch**	**Bmiss**
	[s]	[%]	[%]	[%]	[%]	[%]	[%]
k-nucleotide	49.4817	+2.1	+10.1	+1.2	+3.3	+0.9	+0.8
fannkuch-redux	48.6900	+1.2	+9.5	+0.2	+0.3	+0.7	+1.9
n-body	11.2550	+2.4	+9.6	+0.6	+0.5	+0.8	+0.7
binary-trees	8.4233	+1.8	+9.9	+1.2	+1.9	+0.8	+2.3
spectral-norm	8.2283	+0.9	+8.9	+0.0	+0.0	+0.0	+0.0
fasta	3.7917	+1.9	+10.5	+1.0	+0.6	+0.7	+0.7
integer	1.4900	+2.1	+11.0	+1.3	+1.3	+1.3	+3.4
exp3_8	1.2600	+2.4	+11.9	+4.4	+1.5	+1.7	+1.3
pidigits	1.0583	+5.7	+16.1	+7.7	+3.9	+3.9	+3.9
wheel-sieve1	0.9200	+1.1	+9.8	0.0	0.0	+0.2	+0.2
reverse-complem	0.8200	+2.4	+9.8	+1.2	+1.2	+2.0	+1.2
kahan	0.8183	+1.0	+9.0	-0.6	-0.2	-0.2	+1.8
sched	0.7000	+2.9	+11.4	+1.4	+1.4	+1.4	+4.3
knights	0.6900	+1.9	+10.1	+1.4	+0.2	+1.4	+1.4
-1 s.d.	—	+1.6	+9.9	+0.7	+0.4	+0.5	+1.1
+1 s.d.	—	+4.6	+14.2	+4.5	+3.4	+4.0	+4.6
Average	—	+3.1	+12.0	+2.5	+1.9	+2.2	+2.8

Figure 6.11: Sampling Mutator Time Overhead (2/2)

	Sample Rate	**Overhead**	**Message Rate**	**Overhead**
	[Sample/s]	[μs/Sample]	[Msg/s]	[μs/Msg]
Time	998.7	2.649	603.3	4.39
Time2	9 980.1	2.169	1 458.1	14.84
Alloc	208 610.3	0.003	1 760.9	0.40
Stack	4 662.4	0.468	4 662.4	0.47
Heap	43 139 691.9	0.001	7.7	7 270.24
Cycle	3 091.4	4.607	1 154.6	12.34
Cycle2	27 557.7	3.278	1 408.5	64.14
Cache	976.6	9.445	562.4	16.40
Cmiss	3 292.4	4.089	193.1	69.74
Branch	975.7	7.149	580.5	12.02
Bmiss	2 428.5	4.491	503.5	21.66

Figure 6.12: Sampling Overhead Overview

manages to collect a high number of samples, especially considering its low overhead.

This means that heap allocation sampling is especially efficient at producing a lot of sampling data without inducing overhead – the perfect combination. To see how other approaches stack up, we additionally calculate an overhead estimate *per sample*. Not unsurprisingly, heap allocation profiling is way ahead of the pack at 0.001 μs per sample. Apart from that we see that most profiling approaches just take a few microseconds per sample, which still makes it realistic to collect significant amounts of sampling data. Especially note that the cost remains relatively stable when we change sampling rate for "Time2" and "Alloc2", suggesting that the rate can be used to regulate overheads.

6.2 Usage Scenario

The main claim that we make about our profiling solution is that it is useful for resolving real-life performance problems. We covered performance and data quality in the last sections, yet while these are necessary for a useful profiling tool, they are by no means sufficient. However, in contrast to these quantifiable characteristics it will be very hard for us to actually prove usefulness: There will always be performance problems that we cannot help with, and probably even programmers that will find our approach to profiling especially hard to work with.

Therefore at best we could claim relative usefulness for a given set of problems and a certain type of user, ideally through a user study. This could give us a proper basis to argue about intuitiveness and usefulness of our user interface. However due to time constraints we will not be able to do this for this thesis. Instead, we opt to get our own hands dirty, and show how analysing a non-trivial real-world example *could* look like.

6.2.1 The Code

We will lift our example from the StackExchange website, where on 13th of March 2012 user "Joey Adams" asked the following question in the code review section [1]:

> "I wrote a string escaping function in C, and I'm trying to rewrite it to Haskell. [...]
>
> Here's a naïve implementation based on Data.ByteString.concatMap:"

```
escape1 :: Word8 → ByteString
escape1 c
  | w2c c == '\\'            = B.replicate 4 (c2w '\\')
  | c >= 32 && c <= 126      = B.singleton c
  | otherwise                = B.pack
      [ c2w '\\', c2w '\\'
      , c2w '0' + ((c `shiftR` 6) .&. 0x7)
      , c2w '0' + ((c `shiftR` 3) .&. 0x7)
      , c2w '0' + (c .&. 0x7) ]

escape :: ByteString → ByteString
escape = B.concatMap escape1

main :: IO ()
main = L.getContents >>= L.putStr .
         L.fromChunks . map escape . L.toChunks
```

Listing 6.3: Original Escaping

We have slightly refactored the code to make it fit into the context of this thesis, but the implementation is identical: We process one single byte at a time into an escaped form. For this example, the code author has handled three cases – basic ASCII characters, backslashes and extended characters.

Despite the author's warning about the code's naïvety, it already uses quite advanced implementation techniques: For example, instead of using Haskell's standard String type to handle the data stream, it employs ByteStrings [Coutts et al., 2007b], which are well known for their speed in dealing with I/O tasks. In fact, the author seems to be well aware that fast streaming is a matter of processing data in chunks that are incrementally read from the disk. Therefore, the escaping process first uses map in order to apply escape to every single chunk, followed by using concatMap in order to process single bytes.

[1] http://codereview.stackexchange.com/questions/9998/optimizing-bytestring-escaping

6.2.2 Profiling

Given the effort, the performance of this code is quite discouraging:

> "I'd expect this to be a few times slower than the C version. Nope, it is 125 times slower than the C version."

Let us therefore compile the program with our debug annotations and profile it:

```
$ ghc Original.hs -O2 -rtsopts -eventlog -g -fsave-core
$ ./Original +RTS -ls -Ey -Ea < Original > /dev/null
$ threadscope Original.eventlog  -d ~/ghc
```

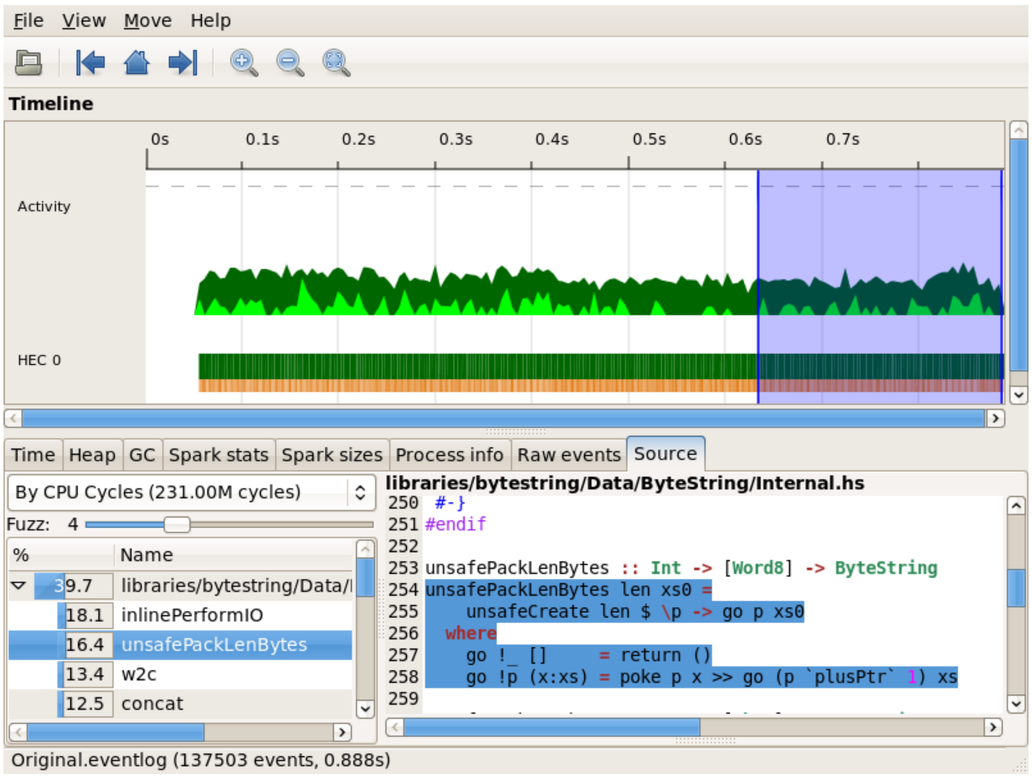
These three lines compile and run the program, and start our analysis tool. Note the command line options: When compiling the program we activate full optimisations, but also pass `-g` in order to request our debug annotations. Due to our work in Section 5.4 on page 129 we are confident that these command line options do not conflict: We can profile a fully optimised program. Furthermore, we requested the optimised Core code to be saved along with the program. This will become useful shortly.

When running our program, we pass it its own executable as text data, and pipe its results into the void. More interesting are the options that we pass to the runtime system (RTS): The −ls option will cause a basic eventlog to be generated, while −Ey and −Ea will cause our profiling infrastructure to collect samples by both cycles as well as allocation. Due to the fact that both have very low skews, there is no strong reason to not combine sampling back-ends like this.

6.2.3 Analysis

If we take a look at the resulting profile view in Figure 6.13 on the next page, it should be clear that we have a pretty tricky performance problem here. The activity graph shows that the program is spending most of its time in garbage collection, and is therefore clearly allocating a lot of data. Unfortunately, even our profiling view turns out to be all over the place: Even with increased fuzz the performance is spread well over a number of modules. We especially cannot associate much performance with the `Original.hs` source file. This suggests that most cost was not found in our code or any portion that could get in-lined into it. Instead, we have apparently pushed most of our complexity into the internals of the bytestring library itself. Given that we would strongly suspect that an efficient implementation using ByteString should be possible, this means that we are probably mis-using the library somehow.

If we further focus on the 40 % CPU cycles spent inside the bytestring library, the picture becomes slightly cleaner. We can easily determine that inlinePerformIO and w2c are basically cost-free, but on the other hand we have two actual "worker" functions that



%	Name
39.7	libraries/bytestring/Data/I
18.1	inlinePerformIO
16.4	unsafePackLenBytes
13.4	w2c
12.5	concat

libraries/bytestring/Data/ByteString/Internal.hs

```
 #-}
#endif

unsafePackLenBytes :: Int -> [Word8] -> ByteString
unsafePackLenBytes len xs0 =
    unsafeCreate len $ \p -> go p xs0
  where
    go !_ []       = return ()
    go !p (x:xs) = poke p x >> go (p `plusPtr` 1) xs
```

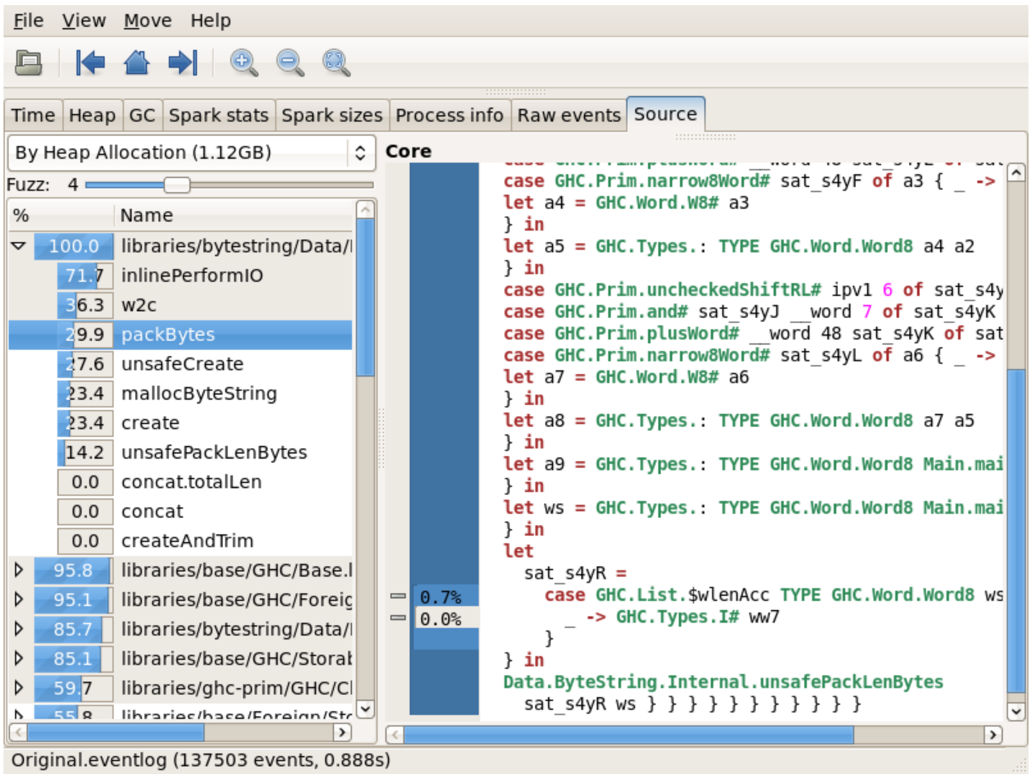
Original.eventlog (137503 events, 0.888s)

Figure 6.13: Original Example Profile

stand out: concat and unsafePackLenBytes. The timeline highlights show us that both get used a fair amount during the execution of our program as explained in Section 5.8.4 on page 171. If we look up their definitions using the source browser, it becomes clear that their function is to pack ByteString objects together.

However, let us take a look at the allocation profile before we attempt to draw any conclusions. For Figure 6.14 on the next page we change the verb, and are now viewing heap allocation statistics. The amount estimation on the left-hand side should be encouraging that we might be on the right track: We gave the program barely 6 MB of input data, yet it managed to burn through more than 1 GB worth of heap allocation while encoding it! Not surprisingly, we can also link 100 % of it with usage of the bytestring library. If we take a closer look at the allocation distribution, we spot that most functions seem to refer to the creation of ByteStrings – such as create or packBytes.

Interestingly enough, these two functions also do not seem to share too much common cost, suggesting that we are dealing with cost coming from different code paths. In fact, if we look at the Core code for packBytes cost, we spot the call to unsafePackLenBytes from above. Furthermore, it is clear why it is so costly: It is allocating a Haskell list (GHC.Types.:), and that within what is apparently the inner loop of the program!


File View Move Help
Time Heap GC Spark stats Spark sizes Process info Raw events Source
By Heap Allocation (1.12GB)
Fuzz: 4

%	Name
100.0	libraries/bytestring/Data/
71.7	inlinePerformIO
36.3	w2c
29.9	packBytes
27.6	unsafeCreate
23.4	mallocByteString
23.4	create
14.2	unsafePackLenBytes
0.0	concat.totalLen
0.0	concat
0.0	createAndTrim
95.8	libraries/base/GHC/Base.
95.1	libraries/base/GHC/Foreig
85.7	libraries/bytestring/Data/
85.1	libraries/base/GHC/Storal
59.7	libraries/ghc-prim/GHC/C

Core

```
case GHC.Prim.narrow8Word# sat_s4yF of a3 { _ ->
let a4 = GHC.Word.W8# a3
} in
let a5 = GHC.Types.: TYPE GHC.Word.Word8 a4 a2
} in
case GHC.Prim.uncheckedShiftRL# ipv1 6 of sat_s4y
case GHC.Prim.and# sat_s4yJ __word 7 of sat_s4yK
case GHC.Prim.plusWord# __word 48 sat_s4yK of sat
case GHC.Prim.narrow8Word# sat_s4yL of a6 { _ ->
let a7 = GHC.Word.W8# a6
} in
let a8 = GHC.Types.: TYPE GHC.Word.Word8 a7 a5
} in
let a9 = GHC.Types.: TYPE GHC.Word.Word8 Main.mai
} in
let ws = GHC.Types.: TYPE GHC.Word.Word8 Main.mai
} in
let
  sat_s4yR =
    case GHC.List.$wlenAcc TYPE GHC.Word.Word8 ws
      _ -> GHC.Types.I# ww7
    }
} in
Data.ByteString.Internal.unsafePackLenBytes
  sat_s4yR ws } } } } } } } } } } }
```

0.7%
0.0%

Original.eventlog (137503 events, 0.888s)

Figure 6.14: Original Allocation Profile

The amount of information that our interface provides might make it hard at first to spot the patterns, but the signs should become clearer: While concat contributed a lot of cost, it contributes almost no allocation, therefore we are apparently concatenating massive numbers of ByteString objects. That we spend a lot of allocation "creating" ByteString objects fits into that picture very well. While all this is going on, we are also clearly spending quite some time constructing lists and reducing them to even more ByteString objects.

If we look at the original program, the problem should now be apparent: We are creating one ByteString object per byte from the original file, which means that over the program run we create roughly 6 million instances. A good number of them even get generated from a list using pack, which is not particularly efficient. The final verdict is simple: We should not try to concatenate millions of tiny ByteStrings – this is not what the library was designed to do.

6.2.4 Blaze-Builder

Fortunately, the author of the original program guessed as much even without going through a detailed performance analysis. In his StackExchange query, he continues:

"Then, I tried using blaze-builder:"

```haskell
escape1 :: Word8 → Builder
escape1 c
  | c == 92                  = fromWrite $ mconcat [bs, bs, bs, bs]
  | c >= 32 && c <= 126      = fromWrite $ writeWord8 c
  | otherwise                = fromWrite $ mconcat
      [ bs, bs
      , writeWord8 $ 48 + ((c `shiftR` 6) .&. 0x7)
      , writeWord8 $ 48 + ((c `shiftR` 3) .&. 0x7)
      , writeWord8 $ 48 + (c .&. 0x7)]
  where bs = writeWord8 92

escape :: ByteString → Builder
escape = B.foldl' f mempty
    where f b c = b `mappend` escape1 c

main :: IO ()
main = L.getContents >>= L.putStr .
   toLazyByteString . mconcat . map escape . L.toChunks
```

Listing 6.4: Blaze-Builder Escaping

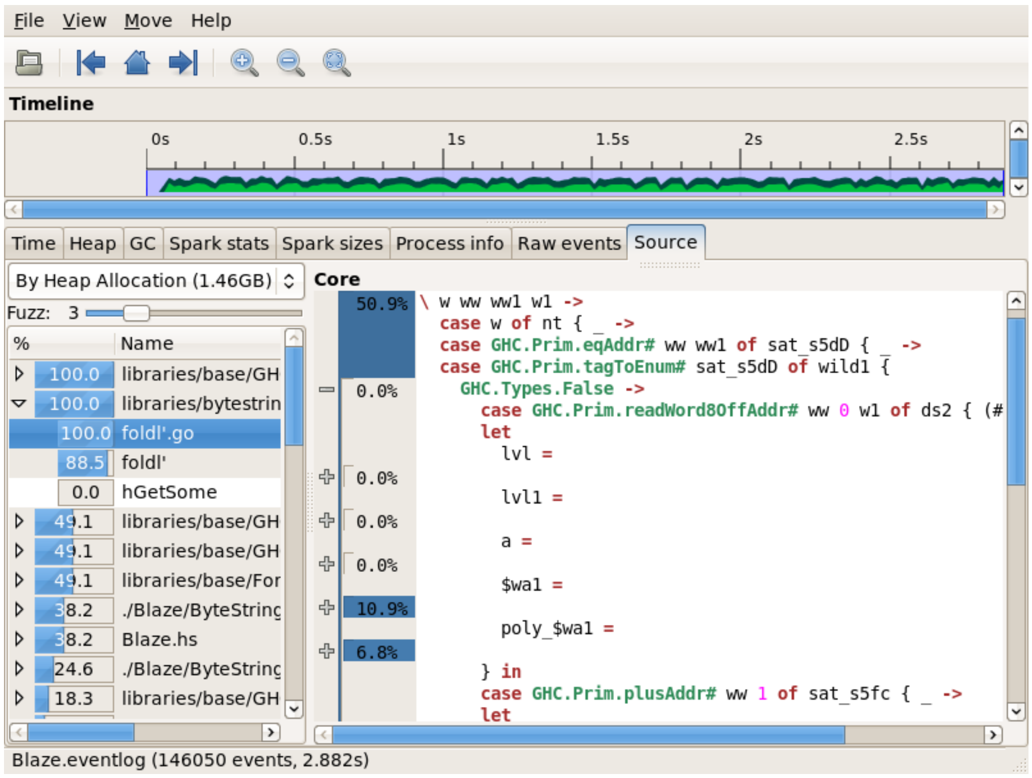
Given our knowledge from the last section, this is a smart choice. In contrast to bytestring, the blaze-builder library addresses exactly the use-case we identified: Building large strings efficiently. In the example, we even see that the author applied the fromWrite optimisation, which basically allows blaze-builder to skip range checks for cases where we statically know how many characters we are going to emit.

However, this version does not fare much better:

> "This made a difference. It's even slower, 300 times slower than the C version. I thought blaze-builder was supposed to be really fast!"

At this point, our user is clearly frustrated. From his point of view, he is doing everything he can in order to allow the Haskell program to perform well, yet it does not seem to help. In fact, this is where he gave up and fell back to using an imperative style.

Fortunately, using our profiling solution we do not have to jump to such conclusions. Figure 6.15 on the following page shows the result of running the new program, using exactly the same parameters as in the last sub-section. A look at ThreadScope's status bar confirms the program author's performance claims: Instead of 0.8 s we are now taking 2.8 s to complete our test run! Also now we are allocating even *more* memory



%	Name
100.0	libraries/base/GH
100.0	libraries/bytestrin
100.0	foldl'.go
88.5	foldl'
0.0	hGetSome
49.1	libraries/base/GH
49.1	libraries/base/GH
49.1	libraries/base/For
38.2	./Blaze/ByteString
38.2	Blaze.hs
24.6	./Blaze/ByteString
18.3	libraries/base/GH

```
50.9%  \ w ww ww1 w1 ->
         case w of nt { _ ->
         case GHC.Prim.eqAddr# ww ww1 of sat_s5dD { _ ->
         case GHC.Prim.tagToEnum# sat_s5dD of wild1 {
 0.0%      GHC.Types.False ->
             case GHC.Prim.readWord8OffAddr# ww 0 w1 of ds2 { (#
             let
               lvl =
 0.0%
               lvl1 =
 0.0%
               a =
 0.0%
               $wa1 =
10.9%
               poly_$wa1 =
 6.8%
             } in
             case GHC.Prim.plusAddr# ww 1 of sat_s5fc { _ ->
             let
```

Blaze.eventlog (146050 events, 2.882s)

Figure 6.15: Blaze-Builder Profile

on the heap, up to 1.46 GB from 1.12 GB we had previously. Somehow, things have gone even more wrong than before. On the other hand, we can now spot a proper "hot spot" in our performance data: Even low amounts of fuzzing allow us to pin 100 % of the allocation on foldl '.go, the worker function of foldl ' from the bytestring library. We actually have a similar kind of problem as our "factorial" example: GHC pulled all important parts into a self-contained inner loop, but has failed to optimise it well.

This makes a good starting point for some Core exploration, so let us have a closer look at Figure 6.15. Core folding as explained in Section 5.8.8 exposes the inner loop from foldl ': The function starts by comparing addresses ww and ww1, continues with reading one byte from ww, followed by increasing the address by 1. If we scroll a bit further down, we find a recursive call. So where exactly is our allocation coming from? In fact, we are looking right at it: The let bindings shown are the main source of allocations, due to the fact that each of them will capture our current address as well as the last read byte. The reason that the let expression is not directly annotated with the 50.9 % allocation is an idiosyncrasy of heap allocation as discussed back in Section 5.7.6 on page 167: The problem is that due to the case expression we can not tell what branch the allocation is coming from. On the other hand, here the False is clearly the "loop" case, so this is where our problem must stem from.

But why are we allocating so many closures in the first place? If we look at the structure of the remainder of the Core code, it looks as follows:

```
go = λbld addr end state → {- ... -}
  let { a = {- ... -}, $wa1 = {- ... -}, {- ... -} } in
  case plusAddr# addr 1 of addr' →
  let bld' = λ step →
        let step' = case byte of
              92 → let step' = λrange s → {- ... -}
                   in step'
              _  → let step' = λrange s → {- ... -}
                   in step'
        in bld step'
  in go bld' addr' end state'
```

Listing 6.5: Blaze-Builder Core

This small forest of lambda expression spawns the rest of the allocation of our program. Note that we renamed variables to reflect their type here, so for example byte is now the byte that was read using readWord8OffAddr# in Figure 6.15.

So what is going on here? Well, we see the recursive call, and we are clearly passing a new builder bld', just as we would expect. However, we might be slightly surprised at how complicated the definition of bld' has turned out: In fact, it is a lambda value that passes our step to the bld closure we got as a parameter! So we are, in fact, passing our step back to the lambda we got from our *left*. This should start sounding awfully familiar at this point: When we "hacked" our factorial implementation to turn a right fold into a left fold, we were also passing continuations between fold iterations. Except that this time, we are apparently turning a left fold into a right fold!

The reason for this whole debacle is of course that the program author apparently had a wrong intuition about what a Builder is in the first place. Here is the slightly simplified definition from the blaze-builder library:

```
newtype Builder
  = Builder (∀ r. BuildStep r → BuildStep r)
newtype BuildStep r
  = BuildStep (BufRange → IO (BuildSignal r))
```

Listing 6.6: Builder definition

As we might have guessed from the Core code, we are actually dealing with a nested function type here. BuildStep encapsulates a primitive write: It takes a buffer position and attempts to write some data into it. The point of Builder is that we string them together using continuation passing style.

We can see roughly how this works by considering the escape1 function from Listing 6.4 on page 195. If we expand the newtype wrappers we obtain:

```
escape1 :: Word8 → (BufRange → IO (BuildSignal a))
           → BufRange → IO (BuildSignal a)
```

Listing 6.7: escape1 type

Which is basically saying "give me a byte and a way to continue writing". The point of passing a continuation is simple: If we find that our content does not actually fit into the current buffer, blaze-builder will have to allocate a new buffer to fill. Constructing Builder like this means that every function naturally receives its buffer from the Builder on its left, therefore this becomes a very low-overhead operation. In the end, this means that for efficient execution we want the following recursive structure:

```
escape1 byte0
  (escape1 byte1
    (escape1 byte2))
  buf
```

Listing 6.8: Recursive escape1

It is not hard to see that this is, in fact, a right fold. Therefore the choice of foldl' has clearly been mistaken.

6.2.5 Folding

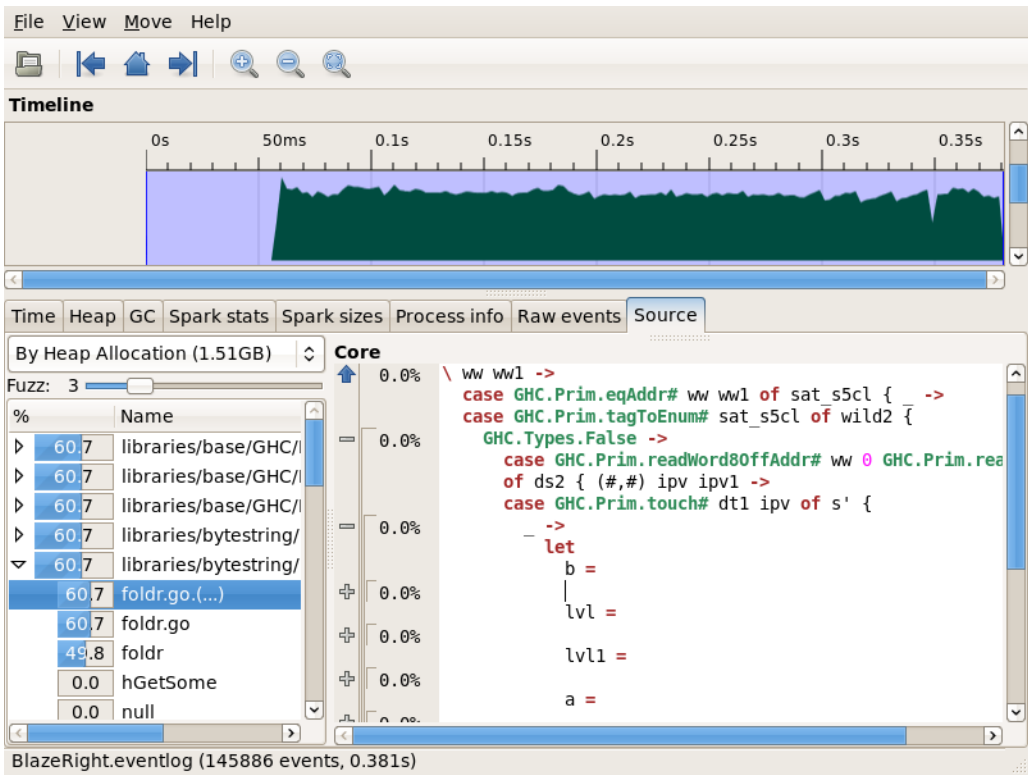
Of course this suggests that we might want to try to use a right fold. So let us simply flip the direction of the fold in the program from Listing 6.4:

```
escapeCopyBytea2 :: ByteString → Builder
escapeCopyBytea2 = B.foldr f mempty
     where f c b = escape1 c `mappend` b
```

Listing 6.9: Right Fold Version

In Figure 6.16 we have opened the profile generated by the resulting program. And clearly we are on the right track: As the status bar shows, just this small change has increased the performance almost 8-fold relative to our first blaze-builder version, and 2-fold over the initial bytestring code. Not bad!

However, a closer look at the profile might lessen our enthusiasm: The new version actually allocates *even more* memory than all our previous versions. This might seem slightly surprising given our good performance, and is a great example for why reasoning about verbs can sometimes be tricky: While allocation always has some overhead, the concrete cost depends on how long we keep references around. And given that we are



```
60.7  libraries/base/GHC/I
60.7  libraries/base/GHC/I
60.7  libraries/base/GHC/I
60.7  libraries/bytestring/
60.7  libraries/bytestring/
  60.7  foldr.go.(...)
  60.7  foldr.go
  49.8  foldr
   0.0  hGetSome
   0.0  null
```

```
0.0%  \ ww ww1 ->
        case GHC.Prim.eqAddr# ww ww1 of sat_s5cl { _ ->
        case GHC.Prim.tagToEnum# sat_s5cl of wild2 {
0.0%      GHC.Types.False ->
            case GHC.Prim.readWord8OffAddr# ww 0 GHC.Prim.rea
            of ds2 { (#,#) ipv ipv1 ->
            case GHC.Prim.touch# dt1 ipv of s' {
0.0%          _ ->
              let
                b =
0.0%            |
                lvl =
0.0%
                lvl1 =
0.0%
                a =
```

BlazeRight.eventlog (145886 events, 0.381s)

Figure 6.16: Blaze-Builder Right-Fold Profile

now walking through the ByteString in the "right" order, this basically means that our allocated objects will die very quickly, leading to low actual overhead.

However, this does not explain why we are allocating so much data in the first place. As we see in Figure 6.16, folding away unrelated Core code exposes a familiar structure: address comparison, followed by reading one byte, followed – again – by a whole bunch of closure allocations. So what is going on this time? If we pull out the Core code structure again, we see that our inner loop has changed to the structure shown in Listing 6.10 on the next page. Note that while in Listing 6.5 we built closures as arguments, this time around the whole function is basically a big nested lambda-term. The expanded type of go even looks remarkably similar to the escape1 type: Given two addresses and a continuation we promise to fill a buffer.

However, note that instead of matching all arguments at the top level, GHC has left lambdas "inside" wherever it could. The worry behind this is the same as we explained back in Section 3.4.6 on page 36: What if our worker function go gets called often with the same two addresses, but different step and range parameters? The allocated closures are basically GHC organising thunks in order to save back work if this should happen to be the case. And apparently, this time around the control flow is too complicated for the arity analysis of Breitner [2004] to figure out that this is a bad idea.

```
go :: Addr#
   → Addr#
   → (BufRange → IO (BuildSignal a))
   → BufRange
   → IO (BuildSignal a)
go = λ addr end →
  {- ... -}
   let {- ... -}
     bld =
       λ step →
         let a = {- ... -}
         in case byte of
              92 → let step' = λrange s → {- ... -}
                   in step'
              _  → let step' = λrange s → {- ... -}
                   in step'
   in bld
```

Listing 6.10: Blaze-Builder Core – Right Fold

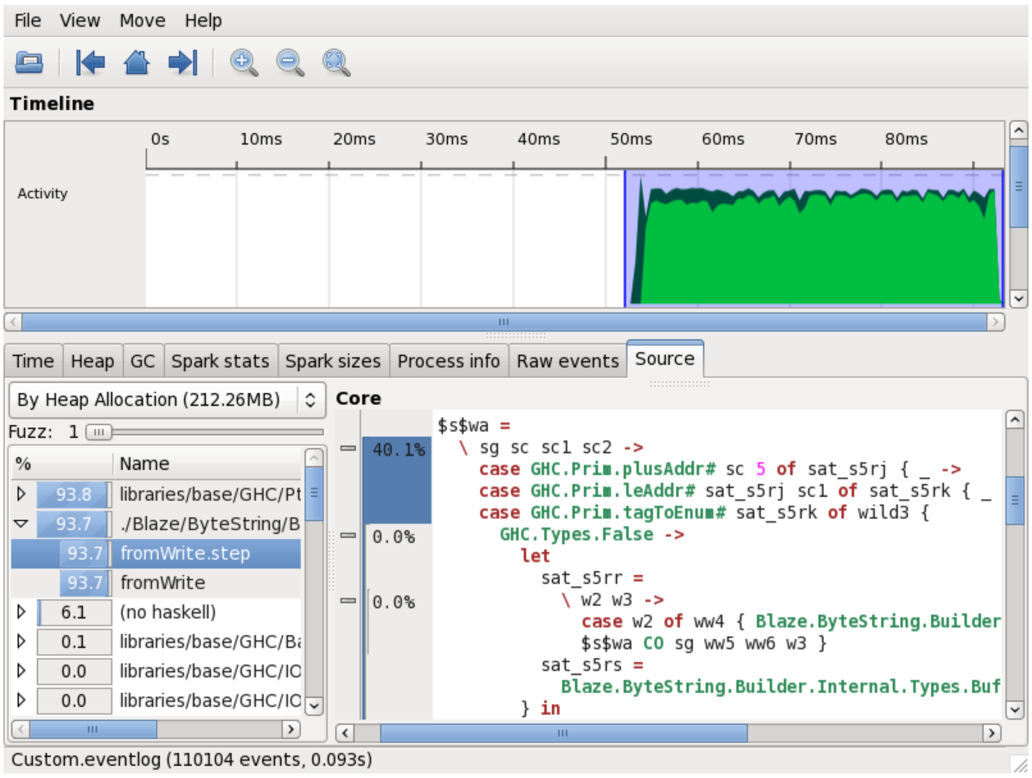
6.2.6 Tailoring

At this point we have run out of easy solutions. The reason that the worker function go ends up getting composed out of three layers of lambda expressions is something that comes naturally from the way we have structured our computation. However, by this point we know exactly what the core of the problem is: We need the inner worker function derived for foldr to receive the right arity so it can perform Builder operations efficiently. To achieve this, we simply take the definitions of foldr from bytestring and mappend and mempty from blaze-builder and "manually" inline them into each other. As we see in Listing 6.11, this means unravelling the internals of both libraries quite a bit. However, this allows us to force GHC to use the arity we want: Now the loop go explicitly iterates over the input buffer (p and q) and the output buffer (range) in parallel.

As Figure 6.17 shows, this relatively straightforward change makes a significant difference. In fact, is has reduced the runtime down to just 93 ms! This is low enough that even the set-up cost of event log generation becomes significant (see Section 5.7.1 on page 162). In fact, without profiling the program now achieves a total run time of around 41 ms, which is a substantial 8-fold improvement over the previous version.

```
foldBuilder :: (Word8 → Builder) → B.ByteString
                → Builder
foldBuilder f (PS fp s l) =
  fromBuildStepCont $ λcont range → do
    let ptr = unsafeForeignPtrToPtr fp
        go !p !q !range'
          | p == q      = cont range'
          | otherwise = do
            touchForeignPtr ptr
            c ← peek p
            let p' = p `plusPtr` 1
                step = BuildStep (go p' q)
            runBuildStep (unBuilder (f c) step)
                         range'
    lgo (ptr `plusPtr` s) (ptr `plusPtr` (s+l)) range
```

Listing 6.11: Specialised Fold



%	Name
93.8	libraries/base/GHC/Pt
93.7	./Blaze/ByteString/B
93.7	fromWrite.step
93.7	fromWrite
6.1	(no haskell)
0.1	libraries/base/GHC/Ba
0.0	libraries/base/GHC/IO
0.0	libraries/base/GHC/IO

```
         $s$wa =
40.1%     \ sg sc sc1 sc2 ->
            case GHC.Prim.plusAddr# sc 5 of sat_s5rj { _ ->
            case GHC.Prim.leAddr# sat_s5rj sc1 of sat_s5rk { _
            case GHC.Prim.tagToEnum# sat_s5rk of wild3 {
0.0%          GHC.Types.False ->
                let
                  sat_s5rr =
0.0%                \ w2 w3 ->
                      case w2 of ww4 { Blaze.ByteString.Builder
                      $s$wa CO sg ww5 ww6 w3 }
                  sat_s5rs =
                    Blaze.ByteString.Builder.Internal.Types.Buf
                } in
```

Custom.eventlog (110104 events, 0.093s)

Figure 6.17: Blaze-Builder Profile – Optimised Fold

6.2.7 Manipulation

At this point, we have almost managed to push the performance of the Haskell version into the same realm as the C implementation, which takes 13 ms to complete the same task. What is more, the Core is now compact enough that it becomes quite easy to spot the source of the remaining inefficiencies. For example, in Figure 6.17 on page 201 we spot that the program is still allocating about 212 MB worth of heap data. This is a lot better than before, but can certainly still be improved. This time, the analysis leads us to where data gets written into the buffer. Here is what this looks like for escaped characters:

```
{- ... determine what to write ... -}
let step :: Addr# → Addr# → State# RealWorld
              → (# State# RealWorld, BuildSignal r #)
      step = λ addr end rw →
        case plusAddr# addr 5# of addr' →
        case leAddr# addr' end of
          False → let cont = λ buf' rw' →
                               case buf' of BufRange addr' limit' →
                                step addr' limit' rw'
                          in (# rw, BufferFull 5 addr cont #)
          True → {- ... write out data ... -}
in step addr end rw
```

Listing 6.12: Blaze-Builder Writing Core

Note that this time the two addresses addr and end refer to the output buffer. We conclude that the point of this code is to check whether enough space is available to output the desired number of bytes – here 5 for a character in escaped form. If we run into the end of the buffer, we return a BufferFull signal, which will invoke the continuation with a fresh buffer.

In this case, the allocation can be shown to originate from the step lambda[2]. In contrast to what one might expect, this is no instance of Let-No-Escape as explained back in Section 3.5.11 on page 54. After all, a reference to writer clearly *can* escape through the cont closure! While we know that this is a very rare event, this still forces GHC to actually allocate the writer closure on the heap, complete with all life variables just in case we need to interrupt and then resume our writing process.

[2]Unfortunately, the code generator plays a trick on us here – the code generated for the step lambda advances the heap pointer in anticipation of allocating cont, only to reset it in the (expected) case that we take the second branch. As it happens, this temporary increase of the heap pointer is what causes all calls into the memory system, therefore "shadowing" the true source of the allocation. Fortunately, it is not far away, so it is not hard to make the connection.

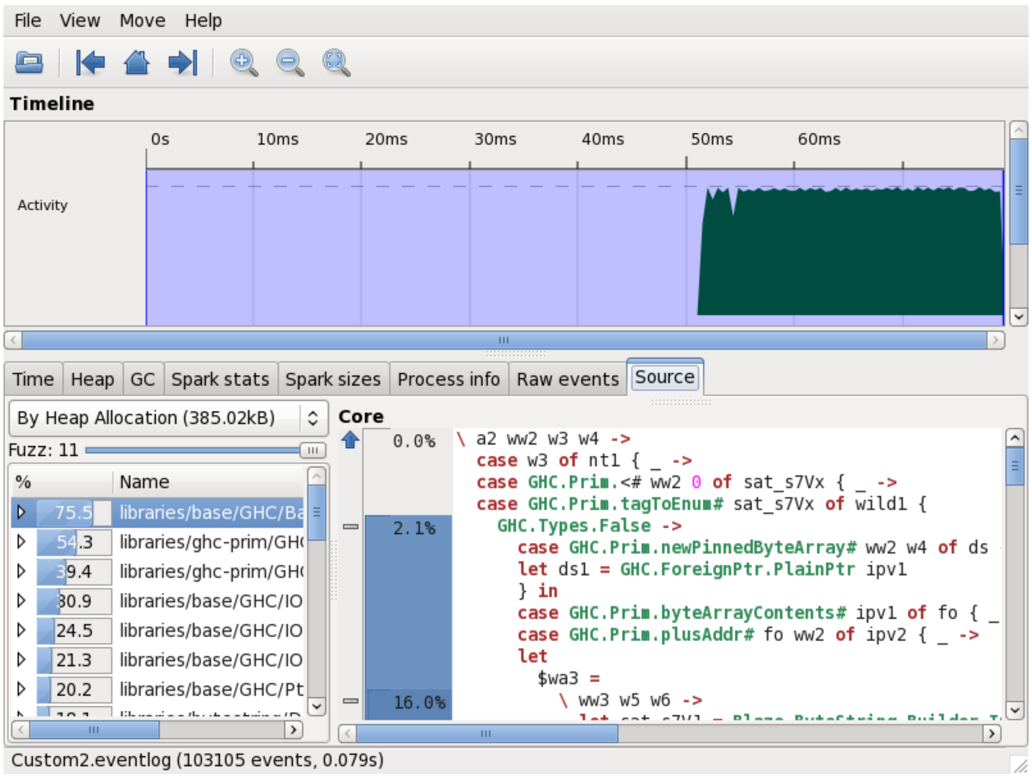
In our example, it would clearly be better if we could just directly invoke the code without the need to wrap it into a closure first. However, this is not as easy as it sounds, as the data to write out comes from either the stack or the closure depending on the code path that led us there. This means that in order to improve the code, we need to duplicate the writing code. Fortunately, in this instance this is quite easy, at least if we are advantageous enough to directly manipulate the implementation of blaze-builder. To be specific, the Core code from Listing 6.12 derives directly from the library's fromWrite function. We can simply adapt this function to look as follows:

```
fromWrite' :: Write → Builder
fromWrite' write = fromBuildStepCont step
  where step k (BufRange op ope)
            | op `plusPtr` getBound write <= ope = do
                op' ← runPoke (getPoke write) op
                let !br' = BufRange op' ope
                k br'
            | otherwise =
                return $ bufferFull (getBound write) op
                                    (step' k)
        step' k (BufRange op ope) = do
          op' ← runPoke (getPoke write) op
          let !br' = BufRange op' ope
          k br'
```

Listing 6.13: Custom fromWrite implementation

We have not changed much, the only difference is the step' function. Originally the library passed a (step k) closure to bufferFull here, which is what led to the Core code shown in Listing 6.12. Now we have set up a specialised code path for resuming the write process, which should allow GHC to either let-no-escape or inline the original step function. Having a separate function also gives us the opportunity to skip the seemingly redundant size check for the fresh buffer.

This change brings the total runtime of the program down to 21 ms, another improvement by a factor of 2. In fact, as Figure 6.18 on the following page shows, the total allocation of the program is now just 385 kB, which is less than the size of the input file. This means that we have eliminated virtually all heap allocation from the inner loop! At this point, we have optimised the program as far as we can. In fact, we have come fairly close to the reference point of 13 ms that the C version took.



%	Name
75.5	libraries/base/GHC/Ba
54.3	libraries/ghc-prim/GH
39.4	libraries/ghc-prim/GH
30.9	libraries/base/GHC/IO
24.5	libraries/base/GHC/IO
21.3	libraries/base/GHC/IO
20.2	libraries/base/GHC/Pt

```
0.0%   \ a2 ww2 w3 w4 ->
         case w3 of nt1 { _ ->
         case GHC.Prim.<# ww2 0 of sat_s7Vx { _ ->
         case GHC.Prim.tagToEnum# sat_s7Vx of wild1 {
           GHC.Types.False ->
2.1%         case GHC.Prim.newPinnedByteArray# ww2 w4 of ds
             let ds1 = GHC.ForeignPtr.PlainPtr ipv1
             } in
             case GHC.Prim.byteArrayContents# ipv1 of fo { _
             case GHC.Prim.plusAddr# fo ww2 of ipv2 { _ ->
             let
               $wa3 =
16.0%            \ ww3 w5 w6 ->
```

Custom2.eventlog (103105 events, 0.079s)

Figure 6.18: Blaze-Builder Profile – Optimised Write

6.3 Wrapping Up

The data we have collected in this chapter is no proof of usefulness, but it gives us an idea how the practice of using our tool might look like. Our case study has shown that we can optimise a Haskell program to the point where it can compete with C in terms of performance. Especially note that while we had to replace certain key parts of the infrastructure, the structure of the code did not change much compared to our first version in Listing 6.4 on page 195! This means that we have indeed managed to improve performance while remaining high-level in our top-level implementation. It is not unrealistic that in future, libraries and compiler optimisations might mature to the point where we can skip some of the "hacks" we had to employ here.

Furthermore, as the first part of the chapter should have shown, we can be quite confident that our tool will work well in a lot of scenarios. We ourselves have tested our tool on a number of programs ranging from small examples to large applications (such as GHC itself), and have found that with careful analysis we can find evidence for most kinds of performance problems. And while such an investigation might admittedly still often require expert knowledge and experience, the added information and guidance makes the optimisation job much easier than it had been before.

Chapter 7

Conclusion

"– We have been torturing you for hours, and it does not even seem to hurt. This will not help you!"

"– Oh yes, it will: It will help to pass the time..."

Asterix the Gaul, René Goscinny & Albert Uderzo

When we started out with this thesis, our claim was that we would be able to provide a novel profiling solution for the programming language Haskell. Our goal was not to displace existing solutions, but to provide new options to users that have struggled with making sense of Haskell's occasionally erratic performance behaviour so far. Our aim was to directly address one of the hardest problems that plagues reasoning about Haskell program performance today: The disconnect between rapidly evolving libraries promising to marry high performance with purely functional programming, and the unfortunate realities of attempting to diagnose the ever more complicated performance problems that this combination can lead to.

7.1 Contributions

In the end, we can claim that we were successful in attacking the profiling problem in a rigorous fashion. As a result, we have found a few novel ways of thinking about the profiling task.

This started in Chapter 2, where we set out to build a foundation for our work. Here is where we defined the terms that we were going to use throughout this thesis. The verb/noun nomenclature we introduce was originally formulated by Irvin [1995], however we believe that the way we have linked it to the causal reasoning process is a fresh interpretation of the concept. Similarly, Chapter 3 mainly serves to introduce existing material, while putting our own spin on existing themes. For example, the performance model we introduce is intended to make stronger claims about the performance of

Haskell programs than usual, as we want to obtain guarantees that we are not going to miss the causal connection of certain verbs in profiling.

The central contributions of this thesis can however be found in Chapter 4. To our knowledge there has been no previous attempt at formally applying causal theory to operational semantics. Our approach of decomposing rule matches into a network of events and determining causality by employing closest-world thought experiments has been specifically developed as a theoretical foundation for this work. The purpose of this is to enable the reasoning about optimisations as shown in Section 4.8: At this point we are able to make well-grounded claims about what the “right” annotation schemes are in every situation. Before the link with causality was established, reasoning at this point was effectively based on worst-case scenarios and consistency concerns. And while the results might not be particularly spectacular, in a language with such tricky semantics as Haskell it is a good idea to have strong guarantees about our claims.

The profiling framework we developed in Chapter 5 forms the second main body of contributions. Apart from the engineering work involved, we believe that our design brings a few new concepts to the table as well. For example, our causality semantics can serve as a baseline for comparing different ways of explaining performance problems. In Section 5.3 on page 121 we used this to evaluate various iterations of cost-centre profiling as introduced by Sansom and Peyton Jones [1995]. Furthermore, our insights into the performance analysis process has led to a novel user interface design tailored towards the specific requirements of working with heavily optimised programs. To be specific, we realised that talking about causal processes involving transformations meant that we can not always determine a singular villain in the source code. Consequently we showed in Section 5.8 on page 168 how we can design the interface so it allows the user to work with such “ambiguous” source code links. To offset the inevitable uncertainty, our Core viewer fills the role of a one-stop analysis tool for unraveling the program’s inner workings. This especially allows us to making sense of exactly how the compiler’s decision gave rise to the program’s performance characteristics. To our knowledge this form of analysis is unprecedented, at minimum within the Haskell world.

7.2 Prior Work

7.2.1 Haskell Profiling

Clearly our work was heavily inspired by the work on previous profiling solutions. To start with, our approach targets the Haskell ecosystem, so our main reference point is obviously the work by Sansom and Peyton Jones [1995], which was recently revisited by Marlow [2012]. The problems of cost attribution led us down similar paths – for example, we argued in Section 4.4.1 on page 72 that laziness is transparent with respects to cause annotations, which has parallels to Sansom and Peyton Jones [1997, Section 3.5]. Furthermore as explicitly explored in Section 5.3.2 on page 122 forwards, we can even map lexical and evaluation scoping to subsets of our full cause terms. However the aim of our works still vary significantly, with our work prioritising the ability to do low-overhead profiling of optimised program over concerns such as cost centre stack consistency.

Next we have to acknowledge the work of Jones et al. [2009] on the eventlog tracing infrastructure as well as the ThreadScope profiling tool, which we extended in order to analyse our data. The main focus of their profiling is reasoning about the behaviour of parallel Haskell programs, which has further improved over the years due to the work of Coutts et al. [2011]. In fact, the format has become something of a lingua franca for profiling parallelism in Haskell variants, with a good number of consumers in the Eden trace viewer [Berthold and Loogen, 2007], HdpHProf [Al Saeed et al., 2012] as well as the recent `ghc-events-analyze` utility [de Vries and Coutts, 2014]. It has even been used for profiling Mercurial programs [Bone and Somogyi, 2011]. And even though profiling parallel programs is out of scope of this work, the high variety of profiling approaches has certainly been an inspiration for developing our general noun/verb nomenclature.

On a similar note, we have to acknowledge work on Haskell profiling that we do not actively touch here. There has been the work of Röjemo and Runciman [1996] on decomposing what we see as the heap residency verb. The work by the same group on coverage analysis [Gill and Runciman, 2007] could also be interpreted as a distant relative to program profiling. Returning to parallel profiling, there have been a number of interesting profiling approaches exploring parallel programs executed using GranSim [Loidl, 1996], such as the work of Hammond et al. [1997]. The approach of Charles and Runciman [1999] to make profile exploration interactive can be seen as having parallels with our work. As Charles [2001] shows, effectively climbing the causal chain backwards is an equally non-trivial pursuit in the parallel context.

7.2.2 General Profiling

Quite obviously Haskell has not been the only language environment that has seen active work on profiling questions. The field is way too large to give a comprehensible literature review. We will therefore mainly focus on instances where existing work inspired our choice of approaching profiling of optimised code using causality theory.

On this account we first have to credit the work of Irvin [1995] which inspired the verb/noun nomenclature used throughout this work, even if it was not interpreted in a causal context. As he notes, verbs and nouns can exist on many different levels of abstractions, which we can "map" in order to obtain more useful views. However, no formal connection to causality is made. The work of Shende [2001] continues this effort of multi-level verb mapping in a distributed context, with their implementation also explicitly addressing the problem of working with compiler optimisations. The more complex verb structure of distributed and parallel systems has also given rise to some work on more complex performance metrics, such as for example in Eyerman and Eeckhout [2008], or the key performance indicators used for assessing performance of cloud computing services [Garg et al., 2013].

Profiling in presence of compiler optimisations has also addressed quite a few times in the past, with for example Appel et al. [1988] already noting that “The machine-code for a function is not necessarily all contiguous. A function may be turned into several pieces of code, with portions of the code for other functions interspersed.”. To our knowledge, Brooks et al. [1992] is the first time that it was considered that the proper way to approach profiling optimised code might be to see it as generated by multiple pieces of source code at the same time. The work of Kaneshiro and Shindo [1996] tackles tracking source code relations on object code moved by program transformations. Finally, it has to be mentioned that the LLVM infrastructure [Lattner and Adve, 2004] as explained in Section 5.6.8 on page 160 also supports somewhat optimisation-resistant debugging annotations via line annotations. Neither work formally approached this problem from a perspective of causal reasoning.

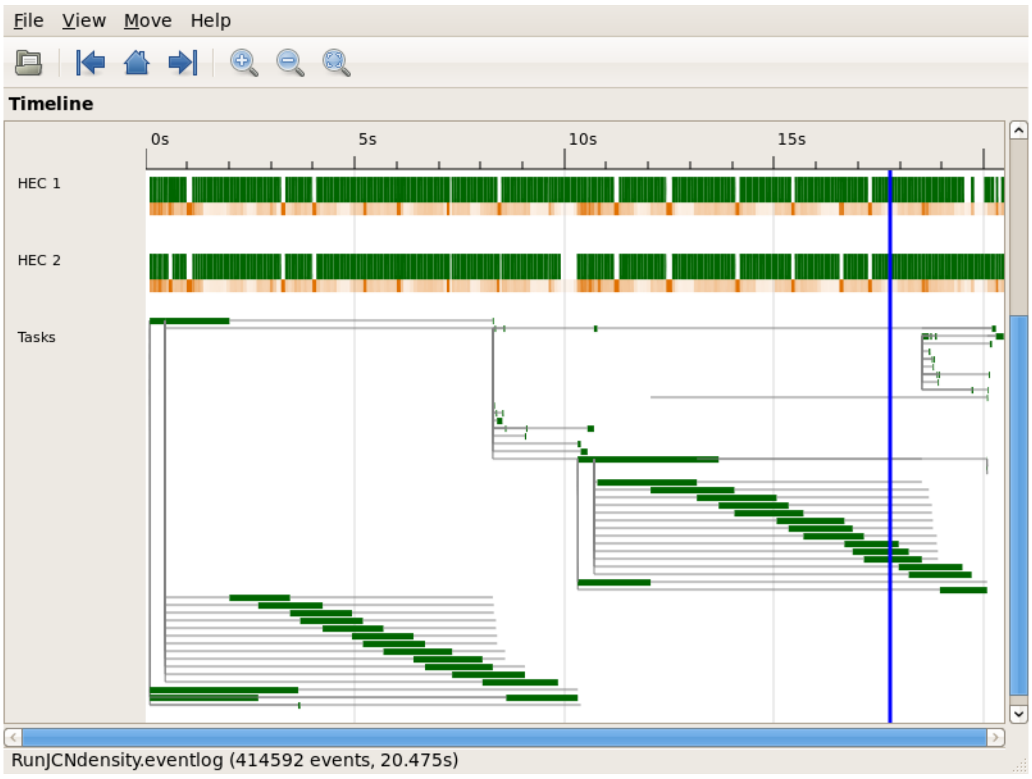
7.3 Future Work

We have made some significant progress on realising our vision for how optimisation-aware low-level profiling could look like for Haskell. On the other hand, it is pretty much the nature of the profiling task that we can never truly claim completeness in any shape or form. In fact, we can easily number fairly straight-forward ways that this work could be otherwise applied or extended.

7.3.1 Parallelism

In this day and age, it seems almost antiquated to focus as much on single-core performance as we have for this thesis. The slow change in hardware architectures means that in future we might have entirely new profiling challenges arising from the need to run programs on parallel architectures. Fortunately, most of our ground work still applies: The user would be facing an abductive reasoning task, and would apply verb and noun abstractions in order to find causal links. The main difference would concern the verb side of things, where we should now be talking about communication delays and idle times.

In fact, in the context of this work we developed a preliminary solution for profiling programs written using the `monad−par` library by Marlow et al. [2011]. The basic idea



RunJCNdensity.eventlog (414592 events, 20.475s)

Figure 7.1: Parallel Profile

here is that the main verb we are trying to track is data locality: Do parallel processes progress on the work load in an orderly fashion, or are we jumping so much that we are likely to incur significant communication delays? This shows that it can sometimes be hard to express verbs in numbers – instead we could use a visual approach as shown in Figure 7.1: Here we arrange the directed job dependency graph of an instrumented monad−par library in a way that highlights the structure of data dependencies, while reflecting how the program executed. In this view, poor locality would show as vertical "jumps", whereas idle times would correspond to horizontal free space. However, we have not had the chance to develop this work much, but still expect the future of parallel profiling to lie in similar concepts.

7.3.2 Technicalities

Furthermore, the amount of technical work that went into this thesis also uncovered a lot of potential for further improvements. One of the most glaring problems with our profiling solution at this point is that unless the compiler helps us by in-lining, we have basically no chance to figure out where a function was called from. While we argue that GHC will do this quite often, this is still a significant blind spot. Fortunately, as we reported in Section 5.3.6 on page 128 it is quite realistic to introduce Haskell stack walking at some point, with Rouhani-Kalleh [2014] and Schröder [2014] already investigating interesting angles of attack.

Furthermore, it is quite unsatisfactory that so far our profiling solution can only target Linux systems. The list of problems ranges from rather trivial problems such as identifying the current memory map (see Section 5.7.1 on page 162) to more involved issues such as how to access hardware performance pointers reliably, as investigated briefly in Section 5.7.4 on page 166. It would be especially nice if we could fully support the LLVM infrastructure as sketched in Section 5.6.8, as this back-end will often generate faster code than the native code generator. On the other hand, this is quite hard to achieve due to the limitations in meta data, as well as apparent data loss during low-level optimisations.

Finally, due to the fact that our profiling approach is mostly based on instruction pointers, it is relatively easy to introduce new verbs. This leaves ample space for future extensions, such as verbs tracking the movement of heap object types through generations. In a similar vain we might attempt to extend our system on the noun side: As Irvin [1995] already notes, it might make sense to track causality for user-defined nouns instead of just source code locations. This could take the form of simple event-log markers identifying program phases as implemented by Coutts et al. [2011], or full-blown noun trees that the program automatically generates in order to explain its own structure. We could for example imagine program interpreters communicating the call stack of the interpreted program to a profiling tool in order to inform analysis.

7.3.3 User Interface

Our extension of the ThreadScope user interface has proved to be a powerful tool for drilling deep into complicated performance problems. However, we could make our approach even more useful if we were better about integrating existing information into our view. For example, there is no direct reason why we could not use our interface to reason about cost-centre profiling data as well – this would allow us to properly reason about stack traces where it is appropriate. From the user's point of view, there is probably little that we would have to change, with even fuzzing as explained in Section 5.8.5 on page 172 having a straightforward equivalent.

Furthermore, one of the unique aspects of our work was that we tried to integrate a view of the intermediate Core language into our profiling view. However, as we noted in Section 5.8.8 on page 176 showing unfiltered Core to the user can be overwhelming, and therefore extra efforts might be warranted. For example, the work on HERMIT [Farmer et al., 2012] shows a number of ways that we can make Core code easier on the eyes and less error-prone to reason about, such as using better variable names than the ones generated by GHC. For the purpose of profiling we would also really like to have basic navigation tools, such as being able to quickly “jump” to a given definition or return to a prior view. Finally, we show Core primarily to allow the user to reason about performance, therefore we could do a better job communicating the performance model to the user. This could be as easy as simply emphasising allocation spots and non-primitive variables. On the other hand we also have quite a few interesting opportunities for code analysis here, which might help identify instances of let-no-escape (see Section 3.5.11 on page 54) or find the amount of live variables of closures.

List of Figures

List of Listings

Bibliography

DWARF debugging information format, version 2. Technical report, UNIX International Programming Languages SIG, 1993. (pages 149, 151)

Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977. (page 40)

Majed Al Saeed, Patrick Maier, Phil Trinder, and Lilia Georgieva. HdpHprof — a profiler for Haskell distributed parallel Haskell. In *The Draft Proceedings of the Symposium on Trends in Functional Programming*, TFP'12, St Andrews, Scotland, June 2012. (pages 162, 207)

David Anderson. libdwarf and dwarfdump, 2011. URL `www.prevanders.net/dwarf.html`. (page 163)

Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25:275–279, June 1987. (page 47)

Andrew W. Appel, Bruce F. Duba, and David B. MacQueen. Profiling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, November 1988. (page 208)

Jost Berthold and Rita Loogen. Visualizing parallel functional program runs: Case studies with the Eden trace viewer. In *Proceedings of the International Conference on Parallel Computing: Architectures, Algorithms and Applications*, PARCO '07, pages 121–128, September 2007. (pages 162, 207)

Paul Bone and Zoltan Somogyi. Profiling parallel Mercury programs with ThreadScope. *CoRR*, abs/1109.1421, 2011. (page 207)

Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*, pages 58–84. Springer Berlin Heidelberg, 1997. (page 27)

Joachim Breitner. Call arity. In *Preproceedings of the 15th Symposium on Trends in Functional Programming*, TFP2014, pages 203–212, May 2004. (pages 16, 38, 39, 199)

Joachim Breitner. dup – Explicit un-sharing in Haskell. *CoRR*, abs/1207.2017, 2012. (page 26)

Gary Brooks, Gilbert J. Hansen, and Steve Simmons. A new approach to debugging optimized code. In *Proceedings of the SIGPLAN conference on programming language design and implementation*, PLDI '92, pages 1–11, 1992. (page 208)

Shirley Browne, Jack Dongarra, Nathan Garner, Goerge Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14:189–204, August 2000. (page 166)

Nathan R. Charles. *Data Model Refinement, Generic Profiling, and Functional Programming*. PhD thesis, University of York, May 2001. (page 207)

Nathan R. Charles and Colin Runciman. An interactive approach to profiling parallel functional programs. In *Implementation of Functional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 650–650. 1999. (page 207)

Chris Clack and Simon L. Peyton Jones. Strictness analysis—a practical approach. In *Proceedings Of a Conference on Functional Programming Languages and Computer Architecture*, pages 35–49, New York, NY, USA, 1985. Springer-Verlag New York, Inc. (page 43)

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on functional programming*, pages 315–326, New York, NY, USA, 2007a. ACM. (page 36)

Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In *Practical Aspects of Declarative Languages 8th International Symposium*, PADL 2007, pages 50–64, January 2007b. (page 191)

Duncan Coutts, Mikolaj Konarski, and Andres Loeh. Spark visualization in ThreadScope. September 2011. URL `haskell.org/haskellwiki/HaskellImplementorsWorkshop/2011/Coutts`. (pages 207, 210)

Edesko de Vries and Duncan Coutts. Performance profiling with ghc-events-analyze, February 2014. URL `www.well-typed.com/blog/86/`. Well-Typed Blog. (pages 162, 207)

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell Compiler. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA, 2009. ACM. (page 27)

Stephane Eranian. Overview of the `perf_event` API. 2009. URL `cscads.rice.edu/workshops/summer09/slides/performance-tools/cscads09-eranian.pdf`. (page 166)

David A. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995. (page 160)

Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3):42–53, 2008. (page 208)

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC Core language programs. *SIGPLAN Notices*, 47(12):1–12, September 2012. ISSN 0362-1340. (page 210)

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM. (pages 43, 135)

Martin Fowler. Yet another optimisation article. *IEEE Software*, 19(3):20–21, May 2002. (page 8)

Saurabh Kumar Garg, Steve Versteeg, and Rajkumar Buyya. A framework for ranking of cloud computing services. *Future Generation Computer Systems*, 29(4):1012–1023, 2013. (page 208)

Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996. ISSN 0164-0925. (page 46)

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM. (pages 25, 26)

Andy Gill and Graham Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009. (pages 33, 176)

Andy Gill and Simon Marlow. Happy: The parser generator for Haskell, 1995. URL `www.haskell.org/happy`. (page 131)

Andy Gill and Colin Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Haskell '07, pages 1–12, 2007. (pages 121, 129, 130, 207)

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge University Press Cambridge, 1989. (pages 27, 141)

Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the symposium on compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM. (pages 11, 122)

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18 (2):109–138, March 1996. (page 29)

Kevin Hammond, Hans-Wolfgang Loidl, and Phil Trinder. Parallel cost centre profiling. In *Proceedings of 1997 Glasgow Workshop on Functional Programming*, 1997. (page 207)

I. Herman, G. Melancon, and M.S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, January 2000. (page 177)

David Himmelstrup. Interactive debugging with GHCi. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 107–107, New York, NY, USA, 2006. ACM. (page 130)

Nathan Howell. Adding Haskell to source languages list. April 2012. URL `dwarfstd.org/ShowIssue.php?issue=120218.1`. (page 150)

Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. ISSN 0956-7968. (page 136)

José Iborra and Simon Marlow. Examine your laziness. Technical Report DSIC-II/09/07, Departamento de Sistemas Informáticos y Computación, Technical University of Valencia, April 2007. (page 130)

R. Bruce Irvin. *Performance measurement tools for high-level parallel programming languages*. PhD thesis, University of Wisconsin, 1995. (pages 9, 205, 207, 210)

Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 81–92, New York, NY, USA, 2009. ACM. (pages 12, 156, 162, 169, 171, 207)

Shaun Kaneshiro and Tatsuya Shindo. Profiling optimized code: A profiling system for an hpf compiler. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 469–473, Washington, DC, USA, 1996. IEEE Computer Society. (page 208)

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on code generation and optimization*, CGO '04, pages 75–86, Washington, DC, USA, March 2004. IEEE Computer Society. (pages 160, 208)

John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, POPL '93, pages 144–154, New York, NY, USA, 1993. ACM. (pages 43, 45, 50)

John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM. (page 23)

David Lewis. *Counterfactuals*. Blackwell Publishers, Oxford, 1973. ISBN 978-0-631-22425-9. (pages 13, 57, 59, 100)

David Lewis. Counterfactual dependence and time's arrow. *Noûs*, 13(4):455–476, 1979. (page 60)

David Lewis. Causation as influence. *Journal of Philosophy*, (97):82–72, 2000. (page 110)

Hai Liu, Neal Glew, Leaf Petersen, and Todd A. Anderson. The Intel labs Haskell research compiler. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 105–116, New York, NY, USA, 2013. ACM. (page 27)

Hans-Wolfgang Loidl. *GranSim's User Guide*. Department of Computing Science, University of Glasgow, 0.03 edition, July 1996. URL `www.dcs.gla.ac.uk/fp/software/gransim/user_toc.html`. (page 207)

Edward S. Lowry and Cleburne W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969. (page 146)

Geoffrey Mainland, Roman Leshchinskiy, and Simon L. Peyton Jones. Exploiting vector instructions with generalized stream fusion. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 37–48, New York, NY, USA, 2013. ACM. (pages 2, 26)

Simon Marlow. Overhaul of infrastructure for profiling, coverage (HPC) and breakpoints. GHC Git repository, November 2011. commit `7bb0447df9a783c`. (page 129)

Simon Marlow. Why can't I get a stack trace? September 2012. URL `haskell.org/haskellwiki/HaskellImplementorsWorkshop/2012`. (pages 125, 135, 135, 206)

Simon Marlow and Simon L. Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the International Symposium on Memory management*, ISMM '11, pages 21–32, 2011. (page 73)

Simon Marlow, Alexey Rodriguez Yakushev, and Simon L. Peyton Jones. Faster laziness using dynamic pointer tagging. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 277–288, 2007. (page 53)

Simon Marlow, Ryan Newton, and Simon L. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82, 2011. (page 208)

Michael McDermott. Redundant causation. *British Journal for the Philosophy of Science*, pages 523–544, 1995. (page 110)

Will Partain. The `nofib` benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993. (page 180)

David A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47:71–75, October 2004. (page 46)

Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, March 2000. ISBN 978-0-521-77362-1. (page 14)

Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(02):127–202, 1992. (pages 28, 36, 42)

Simon L. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS Press, 2001. (page 124)

Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003. ISBN 978-0-521-82614-3. (page 21)

Simon L. Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 327–337, 2007. (pages 176, 185)

Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on*

Functional programming languages and computer architecture, pages 636–666, 1991. (page 34)

Simon L. Peyton Jones and David Lester. *Implementing functional languages: A tutorial.* Prentice Hall, 1992. (page 27)

Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell compiler inliner. *Journal of Functional Programming*, 12(5):393–434, July 2002. (pages 35, 137)

Simon L. Peyton Jones and André Luís de Medeiros Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3 – 47, September 1998. 6th European Symposium on Programming. (pages 40, 41)

Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM. (page 124)

Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference*, JFIT '93, March 1993. (page 27)

Simon L. Peyton Jones, Will Partain, and André Luís de Medeiros Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN international conference on functional programming*, ICFP '96, pages 1–12, New York, NY, USA, 1996. ACM. (pages 33, 85, 104)

Simon L. Peyton Jones, Thomas Nordin, and Dino Oliva. C--: A portable assembly language. In *Implementation of Functional Languages*, pages 1–19. Springer, 1998. (pages 29, 143)

Simon L. Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, Haskell '01, pages 203–233, September 2001. (pages 30, 112)

Norman Ramsey, João Dias, and Simon L. Peyton Jones. Hoopl: a modular, reusable library for dataflow analysis and transformation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 121–134, New York, NY, USA, 2010. ACM. (page 143)

Michael Riepe. libelf, 2009. URL `www.mr511.de/software`. (page 163)

Niklas Röjemo and Colin Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 34–41, 1996. (pages 120, 207)

Arash Rouhani-Kalleh. Stack traces in Haskell. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, March 2014. (pages 128, 128, 210)

Fritz Ruehr. The evolution of a Haskell programmer, August 2001. URL `www.willamette.edu/~fruehr/haskell/evolution.html`. (page 36)

Patrick M. Sansom. *Execution Profiling for Non-strict Functional Languages*. PhD thesis, University of Glasgow, 1994. (pages 12, 122, 123, 127, 130, 132)

Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 106–116, New York, NY, USA, 1993. ACM. (pages 47, 120)

Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 355–366, 1995. (pages 122, 126, 168, 206, 206)

Patrick M. Sansom and Simon L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, March 1997. (pages 42, 207)

André Luís de Medeiros Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, July 1995. (pages 27, 42)

Tim C. Schröder. Hacking GHC's stack for fun and profit, January 2014. URL `github.com/blitzcode/ghc-stack`. GitHub repository. (pages 128, 210)

Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(03):231–264, 1997. (pages 43, 51, 75, 86, 89)

Sameer S. Shende. *The role of instrumentation and mapping in performance measurement*. PhD thesis, University of Oregon, August 2001. AAI3024533. (page 207)

Don Stewart. Smoking fast Haskell code using GHC's new LLVM codegen, February 2010. URL `donsbot.wordpress.com/2010/02/21/smoking-fast-haskell-code-using-ghcs-new-llvm-codegen`. (page 41)

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 80–91, New York, NY, USA, 2004. ACM. (page 29)

Chris Taylor. *A Formal Logical Analysis of Causal Relations*. PhD thesis, University of Sussex, 1993. (pages 14, 59, 60, 73)

David A. Terei and Manuel M.T. Chakravarty. An LLVM backend for GHC. In *Proceedings of the third ACM Haskell symposium*, Haskell '10, pages 109–120, New York, NY, USA, 2010. ACM. (pages 41, 160)

Johan Tibell. State of Haskell, 2011 Survey, August 2011. URL `blog.johantibell.com/2011/08/results-from-state-of-haskell-2011.html`. (pages 2, 26)

Philip Wadler. The essence of functional programming. In *Proceedings of the 19th symposium on principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM. (page 124)