Semantic Constraints

Daniel Nikpayuk

May 31, 2016

overview

Function composition in mathematics is straightforward and taken for granted, but in application to *functional* programming languages in the realm of computing science and the tech industry it often gets cluttered and becomes bogged down with messy details, leading to bugs and inefficiencies.

Languages like *Haskell* ($\gg=$) have sprung up to address such issues—relying on more "complete" mathematical theory of type deduction systems and the likes—to prevent, reduce, detect, and correct bugs of all varieties. Regardless, there is one area of analysis I have not seen in my studies, and though it may in fact exist (it could be my own short-coming) I have nonetheless decided to explore it on my own here.

In particular, it has to do with analyzing a function and partitioning its *typed* input into their respective subgroups so as to dispatch to the most appropriate subroutine. This is a common enough strategy in coding, and in life: By observing such rigour in ones implementation we end up producing safer code for general reuse. What happens when we compose two such safe functions? How do we best reappropriate our implicit constraints?

left shifting

Prototypical examples are best, so I've chosen the *left shift* operator common to C and many similar such languages:

$$x \ll n$$

If you've been living in a cave and don't know what this is, it takes a string of bits x and shifts them left-wise by the given amount n. As example:

$$0100\ 1100\ 1100\ 0000 \quad = \quad 0000\ 1010\ 0110\ 0110 \ll 5$$

So, we're given this function, which is known to work on the range of values $0 \le n < \ell$ (where ℓ is the register length; 64 bits in my current laptop). But if our input types are integers for both x, n, what happens when we go out of that range? Often enough design specifications leave such instances undefined putting the burden on whichever implementation, but that doesn't exactly help us, especially if we want to make portable code which we know will behave the same across hardware architectures and compiler implementations.

the analysis

So here I will perform a basic analysis of type constraints which will allow us to build a *wrapper* around our unsafe (but efficient) left shift operator. As we are assuming an integer type as input, we start by looking at the arguments: x, n. In particular we are looking for *edge* cases as our starting point.

With our first argument x, shifting any given value by any amount seems reasonable, so there are no edge cases in terms of invalid input (which we would dispatch gracefully to an error or another custom behaviour), but we do have an *efficiency* edge case: $x = 0$. In this case, there's no point in computationally shifting such bits as the result is always the same, so we can streamline the process. In this way we may partition our argument x integer type accordingly:

$$\{\ x < 0, \quad x = 0, \quad x > 0\ \}$$

We perform a similar analysis for our variable n:

$$\{\ n \le 0, \quad 0 < n < \ell, \quad n \ge \ell\ \}$$

this partition is informed by the previously mentioned working range of $0 \le n < \ell$. If $n = 0$, our operator will always return x. If $n < 0$ our operator has undefined behaviour, but my own safe implementation lumps these two possibilities together as $n \le 0$ and expects a return of x either way. Seems reasonable... of course you're free to disagree. The other case $n \ge \ell$ is computationally possible, but as such a range will always return 0 we might as well streamline. I believe the fancy term is *memoize*?

Okay, so we now have our argument constraints. Combined (as a product) this is:

$$X \times N \quad = \quad \{\, x < 0, \quad x = 0, \quad x > 0 \,\} \times \{\, n \le 0, \quad 0 < n < \ell, \quad n \ge \ell \,\}$$

Representing this differently, we have the relation, we have the combinatorial space:

$$\begin{array}{r|ll}
x < 0 & & \\
 & n \le 0 & \to x \\
 & 0 < n < \ell & \to \lambda \\
 & n \ge \ell & \to 0 \\
x = 0 & & \\
 & n \le 0 & \to x \\
 & 0 < n < \ell & \to 0 \\
 & n \ge \ell & \to 0 \\
x > 0 & & \\
 & n \le 0 & \to x \\
 & 0 < n < \ell & \to \lambda \\
 & n \ge \ell & \to 0
\end{array}$$

here, the "to" operator ($\to$) indicates our default return value. The λ is borrowed from the *lambda* calculus to mean we call the operator of interest, in this case, the unsafe left shift $\lambda = (\ll)$.

This above "tree structure" is mathematically special in that it is a *complete dispatch*. The predicate *cases* perfectly partition our function's domain $X \times N$. We can additionally view this dispatch as a *map* data structure, where we look up a given predicate constraint and find the return of interest.

> *The thing is, you have to understand, and this is why I've written this article, is that I see this tree data structure as a mathematically significant* type *all on its own, yet I don't see it represented in programming languages.*

Semantically, its value is actually exactly that: *semantics.* Everything I've studied about language and linguistics says to me it is an abstract semantic *construct.* A semantic *unit.* The exact dispatch returns actual values, in this case either constants or a function to compute, but if you abstract it from that context, not caring about what it returns, it is a higher concept. It is a *filter.*

What does this mean in practice? Such a construct and its implied paradigm privileges *narrative* algorithms. In software engineering and design we modularize, that's what this is. We're modularizing the dispatch constraint out of its respective function. The value in this is it both conceptually as well as computationally separates a "work-horse" algorithm from case dispatching. When you think up an algorithm you'd like to have, you don't think of various edge cases, instead you have an intuitive feel for how it should work for its representative domain, and you code for that. Edge cases are an after-thought. Algorithms specialized for certain "demographics" tend to be quite efficient because they have far less overhead, and in code, they tend to be highly compressed. Win win.

As for this abstract dispatch structure, why is it of value? It's ideal in many ways for the purposes of programming. It's highly rearrangeable to start, this of course having to do with the flexible nature of identity manipulation within symbolic logic (the commutative and distributive laws hold for ($\wedge$) and ($\vee$)). For example, we can take the above dispatch tree and commute its variable priorities, instead of testing against x first then n, we can invert:

$$\begin{array}{r|ll}
n \le 0 & & \\
 & x < 0 & \to x \\
 & x = 0 & \to x \\
 & x > 0 & \to x \\
0 < n < \ell & & \\
 & x < 0 & \to \lambda \\
 & x = 0 & \to 0 \\
 & x > 0 & \to \lambda \\
n \ge \ell & & \\
 & x < 0 & \to 0 \\
 & x = 0 & \to 0 \\
 & x > 0 & \to 0
\end{array}$$

Or, again starting with our original tree, because the distributive law holds, it means we can also refactor:

$$\begin{array}{r|ll}
x < 0 \lor x > 0 & & \\
 & n \leq 0 & \to x \\
 & 0 < n < \ell & \to \lambda \\
 & n \geq \ell & \to 0 \\
x = 0 & & \\
 & n \leq 0 & \to x \\
 & 0 < n < \ell & \to 0 \\
 & n \geq \ell & \to 0
\end{array}$$

The whole point of computation, and computational design in the first place is to *compress*: To do more by saying less. Refactoring is our compression. That is the name of the game. With this semantic construct, we have a lot of flexibility to do just that.

As for compressing this table, let's clear through any illusions and get straight to the best approach: **Compress by return.** Our commuted table above (the one which privileges n over x) is already heading in that direction:

$$\begin{array}{r|ll}
n \leq 0 & & \\
 & x < 0 & \to x \\
 & x = 0 & \to x \\
 & x > 0 & \to x \\
0 < n < \ell & & \\
 & x < 0 & \to \lambda \\
 & x = 0 & \to 0 \\
 & x > 0 & \to \lambda \\
n \geq \ell & & \\
 & x < 0 & \to 0 \\
 & x = 0 & \to 0 \\
 & x > 0 & \to 0
\end{array}$$

So we first refactor:

$$\begin{array}{r|ll}
n \leq 0 \quad \land \quad [x < 0 \lor x = 0 \lor x > 0] & & \to x \\
0 < n < \ell & & \\
 & x < 0 & \to \lambda \\
 & x = 0 & \to 0 \\
 & x > 0 & \to \lambda \\
n \geq \ell \quad \land \quad [x < 0 \lor x = 0 \lor x > 0] & & \to 0
\end{array}$$

Which is pretty ugly to look at, but notice our $[x < 0 \lor x = 0 \lor x > 0]$ actually covers the whole domain of x and so our tree simplifies to:

$$\begin{array}{r|ll}
n \leq 0 & & \to x \\
0 < n < \ell & & \\
 & x < 0 & \to \lambda \\
 & x = 0 & \to 0 \\
 & x > 0 & \to \lambda \\
n \geq \ell & & \to 0
\end{array}$$

It's that pesky $0 < n < \ell$ case which prevents a clean compression!

$$\begin{array}{r|ll}
0 < n < \ell & & \\
 & x < 0 & \to \lambda \\
 & x = 0 & \to 0 \\
 & x > 0 & \to \lambda
\end{array}$$

complexity. . . *you know what I'm saying?*

We can still rearrange:

$$\begin{array}{r|ll} 0 < n < \ell & & \\ & x = 0 & \to 0 \\ 0 < n < \ell & & \\ & x < 0 & \to \lambda \\ & x > 0 & \to \lambda \end{array}$$

and refactor:

$$\begin{array}{rcl|l} 0 < n < \ell & \wedge & x = 0 & \to 0 \\ 0 < n < \ell & \wedge & [x < 0 \vee x > 0] & \to \lambda \end{array}$$

plugging this back into our table:

$$\begin{array}{rcl|l} n \leq 0 & & & \to x \\ 0 < n < \ell & \wedge & x = 0 & \to 0 \\ 0 < n < \ell & \wedge & [x < 0 \vee x > 0] & \to \lambda \\ n \geq \ell & & & \to 0 \end{array}$$

and so by simplifying just a little further we end up with a table compressed entirely by dispatch return:

$$\begin{array}{rcl|l} n \leq 0 & & & \to x \\ n > 0 & \wedge & [n \geq \ell \vee x = 0] & \to 0 \\ 0 < n < \ell & \wedge & [x < 0 \vee x > 0] & \to \lambda \end{array}$$

So there we have it. We have compressed our semantic unit *by return.* It remains complete as it still partitions the whole of our expected function domain. If you're an optimization happy engineer, you might notice that in practice we can simplify this further:

$$\begin{array}{rcl|l} n \leq 0 & & & \to x \\ n \geq \ell & \vee & x = 0 & \to 0 \\ \textit{otherwise} & & & \to \lambda \end{array}$$

Notice our $n > 0$ condition is gone in the second row predicate? And how our third row doesn't even test and just calls the function? The reason being is if we were going through the cases with "if then else" conditionals (or some syntactic sugar thereof), by failing the first predicate it means we logically know $n > 0$ and so we can simplify the second predicate. We also know that by failing the first two predicates, all other possibilities are still covered—it's still a complete partition—and so we don't actually need to test any further, we can save ourselves that effort.

If you understood and enjoyed this optimization trick, feel free to pat yourself on the back, and in actual implementation I plan to do this in my own code for this exact function, but within a larger systemic logic, this is bad design. The reason being is that up until now we have been performing *lossless* semantic compression, but this clever trick is now in the realm of *lossy* compression. This trick works for this exact ordering of predicates, but if ever you want to reuse this semantic filter, it no longer is as flexible to reshape as it was before the lossy conversion took place. It's far less reusable now.

abstraction

So far, to be fair, I haven't actually done anything in this article but work with database tables and logic queries, so you might say the theory I'm looking for is database theory. I'm pretty sure there's an overlap, but the programming language I envision working with would have native support for filters more general than predicate logic. There are afterall many applications where the partitioning of a type domain isn't so ideal as the integers and their respective interval arithmetic.

Okay, so we've gotten use to this idea of a semantic dispatch, and we know how to analyse a computational function to generate its associated dispatch, but if such dispatches were their own *type*, how would we use them?

As an object, it is basically a partition of a given set which has structure, but is fluid in how we can reshape it. If you have two objects of this type, what do you do with them? Are there any binary operators? To be honest, as for binary operators specifically, I don't know. I do have a natural application in mind we can abstract from though: *function composition.*

Let's say you construct a function as the composition of two other function:

$$h := g \circ f : A \to C$$

where $f : A \to B$, $g : B \to C$. In practice, $h : A \to C$ is a specification, while $g \circ f$ is its implementation. As such we have to verify the specification and implementation match up, otherwise we get bugs.

Let's pretend we've abstracted a *dispatch* semantic from h, our edge cases being determined by the specification and its intuitive meaning. We'll call this dispatch $D(h)$. We'll keep things simple and not worry about any underlying structure of $D(h)$ and only worry about the fact that it partitions our function domain $A = \cup\{D_j(h)\}_j$. So there's some index k and an accompanying subset $D_k(h) \subseteq A$ which is intended to dispatch to our implementation $g \circ f$. As we've already implemented, we should also know its dispatch: $D(g \circ f)$.

So our constructed function works if $D_k(h)$ is a subset of the domain of $g \circ f$, otherwise we have to split the difference, creating new cases for our root dispatch $D(h)$. In such a situation, we are left with a new dispatch $D'(h)$. Our interest at this point is in the fact that $D(h)$ was assumed as compressed as possible (local maxima if a global maximum compression doesn't exist), but there's not guarantee about $D'(h)$. Basically, if you recall we compressed by return, so we would need to extend our compression if one or more of the new return cases generated by synchronizing $D_k(h)$ and $D(g \circ f)$ was already a return case of of $D(h)$. With that said, that's the general routine for this sort of thing. It wouldn't be too difficult to "update" the semantic constraints every time you compose new functions to retain safety and efficiency.

conclusion

So what's the value of all this? Part of it is *efficiency*. If you only code safe functions, then when you compose them but don't recompress the new dispatch, you'll end up doubling up—even more so the more you compose. You'd be testing the same case more than once—by the fact that they overlap because of the implicit nesting—before accessing the part of the function that did the real work. Such overhead does add up. It's like having to pay a toll for every bridge you cross. But to be fair, even this is reasonably low-cost overhead given that it's boolean conditional testing, so is it worth the effort? In some cases, yeah!

For me it would also be nice if this could actually be automated (which it can, at least partially). It would speed up the design and creation of safe reusable function code. I doubt this automation would be compiler-wide, because the ability to recompress is very much dependent on the domain type and what kind of expressive sub-language we have available to express partitions. With the integers we have interval notation—very expressive! But we can't assume all arbitrary types have such convenient notation (aside from generic set-theoretic notation).

What else?

Aside from efficiency and possible automation, I think it's a coder's best practice to get into the habit of this sort of analysis, or at least be comfortable in knowing how. Why? As it stands, there are a lot of code functions in the real world being defined *wastefully*. By modularizing the work-horse part of a function—the pure clean compressible work-horse code—from the dispatch code, we have cleaner design using fewer parts. This builds better code.

Anyway, that's all I can think of for now. If you know of this topic already in the literature, please share, it's fun to explore and it improves my own ability, but I don't always have time to reinvent each and every wheel. Thanks.