Abstract. The well-known third list homomorphism theorem states that if a function $h$ is both an instance of $\text{foldr}$ and $\text{foldl}$, it is a list homomorphism. Plenty of previous works devoted to constructing list homomorphisms, however, overlook the fact that proving $h$ is both a $\text{foldr}$ and a $\text{foldl}$ is often the hardest part which, once done, already provides a useful hint about what the resulting list homomorphism could be. In this paper we propose a new approach: to construct a possible candidate of the associative operator and, at the same time, to transform a proof that $h$ is both a $\text{foldr}$ and a $\text{foldl}$ to a proof that $h$ is a list homomorphism. The effort constructing the proof is thus not wasted, and the resulting program is guaranteed to be correct.

1 Introduction

A function $h$ on lists is called a list homomorphism \cite{1} if it satisfies

$$h(xs + ys) = h(xs) \circ h(ys),$$

(1)

for some associative operator ($\circ$). We wish to identify list homomorphisms due to potential chances of parallelisation: to compute $h$, one may arbitrarily split the input list into $xs + ys$, compute $h(xs)$ and $h(ys)$ in parallel, and combine the results using ($\circ$).

The well-known third list-homomorphism theorem \cite{7} says that a function is a list homomorphism if it can be described as an instance of both $\text{foldr}$ and $\text{foldl}$. That is, there exists ($\circ$) satisfying (1) if

$$h = \text{foldr} \left(\langle\right) e = \text{foldl} \left(\Rightarrow\right) e,$$

(2)

for some ($\langle$) and ($\Rightarrow$). For example, $\text{sum} = \text{foldr} \left(\langle\right) 0 = \text{foldl} \left(\Rightarrow\right) 0$ and, indeed, there exists an ($\circ$) such that $\text{sum}(xs + ys) = \text{sum} xs \circ \text{sum} ys$ — for this simple example, ($\circ$) happens to be ($\Rightarrow$) as well. The proof presented by Gibbons \cite{7} showed that (1) can be satisfied by picking $x \circ y = h \left(\langle x^{-1} + y^{-1}\right)$, where $h^{-1}$ is a weak inverse of $h$, that is, one such that $h^{-1}(h(x)) = x$, which always exists if we assume a set-theoretic semantics.

One naturally wonders whether list homomorphisms can be mechanically constructed. Hu et al. \cite{8} proposed to construct list homomorphisms by fusion with existing ones. Geser and Gorlatch \cite{6} applied term rewriting techniques to construct a definition of ($\circ$) from that of ($\langle$) and ($\Rightarrow$). More recent developments attempt to apply the third list-homomorphism theorem to mechanical construction.
of list homomorphisms. Morita et al. [10] proposed to automatically construct \((\odot)\) by picking some weak inverse \(h^{-1}\) and simplifying \(h (h^{-1} x + h^{-1} y)\). For \(\text{sum}\), one might pick \(\text{sum}^{-1} x = [x]\), and the system simplifies \(\text{sum} (\text{sum}^{-1} x \oplus \text{sum}^{-1} y)\) to \(x + y\). For the method to work, it is preferred that \(h^{-1}\) has a simple, non-recursive definition, such that \(h (h^{-1} x + h^{-1} y)\) can be easily simplified. The method may even be generalised to trees [9].

Elegant as the approach is, when put into practice, however, one cannot help feeling that we have been solving the wrong problem. In all but the most simple cases, efforts are needed to prove \((2)\), that the \(\text{foldr}\) and \(\text{foldl}\) definitions of \(h\) do define the same function. It occurs often that one of \(<\) or \(>\) is picked as the definition of \(h\), while the other is much harder to find. If the two definitions coincide so obviously that a proof is not necessary, like in the case of \(\text{sum}\), the choice of \((\odot)\) is often equally trivial that a calculation/proof would be merely stating the obvious.

Once we have both \(<\) and \(>\), the operator \((\odot)\) can often be constructed in an ad-hoc but effective manner: we may have a fairly good guess of \((\odot)\) by mixing fragments of code of \(<\) and \(>\). We are still left with proving \((1)\), but the proof often turns out to be very similar to that of \((2)\). For a number of examples we fail to see the weak inverse approach applicable: we may have \((\odot)\) constructed, but cannot find any simple \(h^{-1}\) that “explains” its discovery.

For such problems, one may turn to the approach of Geser and Gorlatch [6]. An inherent disadvantage of term rewriting, however, is the lack of semantic concerns — having produced some \((\odot)\) offers no direct guarantee that it is correct. One would still like to have a proof of \((1)\).

The way to go, we propose, is to transform the proof of \((2)\), which we have to provide anyway, to a proof of \((1)\), after assembling a possible definition of \((\odot)\) from that of \(<\) and \(>\).

\textit{Program Construction: Syntax v.s. Semantics.} In program calculation one transforms a problem specification to a program through algebraic manipulation, thereby guaranteeing its correctness. The program and its correctness proof are developed at the same time.

During the process one is often encouraged to think formally, that is, to think in terms of the syntax rather than the semantics. Rather than focusing on the particular problem domain, the development of the program is ideally driven by syntactical guidelines such as achieving symmetry of expressions, matching the expression against certain forms, and exploiting algebraic properties such as a fusion theorem or the associativity of certain operators. The wish is to relieve programmers of the burden of the complexity in the problem domain through syntactical means — just like how we manipulate arithmetic expressions, using well-designed algebraic rules, without thinking what they “mean.” As the slogan says, “let the symbols do the work” [3].

One of the main aims of researchers is therefore to develop convenient notations and theorems that apply generically to a wide range of problems. Such a style, however, could unfortunately have an unhealthy effect when taken to the extreme. Plenty of works on program calculation claim to have discovered