Recent Applications of Parametricity

Janis Voigtländer

Technische Universität Dresden
01062 Dresden, Germany
janis.voigtlaender@acm.org

1 Introduction

Types play an increasingly important role in program construction, software engineering, and reasoning about programs. They serve as documentation of functionality, even as partial specifications, and can help to rule out whole classes of errors before a program is ever run. Types thus provide qualitative guarantees and enable safe reuse of code components. In particular, static type checking constitutes a limited form of automatic correctness proof below the threshold of full, and undecidable, program verification.

An important asset for maintaining a strong type discipline, which attempts to prevent the use of code in unfit contexts by assigning types that are as precise and descriptive as possible, without forgoing the goal of also flexible reuse is the availability of *polymorphism*, first identified as a distinct concept by Strachey [29]. A polymorphic type, or type scheme, expresses that a certain functionality is offered for different concrete type instantiations, and that in a controlled fashion.

Even though the prospects of strong typing as a lightweight yet powerful formal method have already begun to influence the design of mainstream programming languages, and in particular Java and C# are embracing ever more sophisticated forms of polymorphism [21,10], the real stronghold of typeful programming is in the area of *functional* programming languages like ML [20] and Haskell [25]. The clear mathematical basis of the functional paradigm makes languages adhering to it particularly amenable to precise analysis and thus allows the formalisation and mastery, both theoretically and practically, of very potent type systems. In fact, one of the defining characteristics of Haskell over the last two decades has been its role as a highly effective laboratory in which to explore, design, and implement advanced type-related ideas [11].

So the forefront of type research is still in the field of functional programming, and it can be argued that this setting is best suited for exploring ways of using types for high-level program construction and reasoning about the behaviour of programs. Specifically, a very nice synergy arises from Haskell's type discipline and Haskell's insistence on being a *pure*, rather than just any other "almost", functional language. The "no compromises" attitude vis-à-vis any impulse to relax the compliance with the mathematical concept of side-effect-free functions contributes to the existence of powerful reasoning techniques that connect the types of functions to those functions' possible observable behaviours. One such

technique is the systematic derivation of statements about program behaviour from (polymorphic) types alone. Originating from Reynolds' [26] characterisation of what it means, abstractly, for a function to be fully polymorphic over some part of its type, this approach has been popularised by Wadler [34] under the slogan of "free theorems". It combines fruitfully with algebraic techniques like equational reasoning.

One prominent application area for free theorems has been, and continues to be, the conception and study of semantics-preserving program transformations that can be used in a compiler to optimise for execution speed [9,30,7, and others]. There are also somewhat surprising applications outside the core area of programming language research as such. We report on such applications in Sections 4 and 5. To set the stage, though, Sections 2 and 3 first give a brief introduction to Haskell, its abstraction facilities, and associated reasoning techniques in general and on the type level.

2 Haskell and its Abstraction Facilities

We begin by briefly illustrating some important Haskell concepts, based on examples. This section is not intended to be a thorough introduction to the language, but rather should serve to recall key ingredients of the overall programming methodology, as well as to clarify Haskell's syntax for readers more familiar with other functional languages. We also highlight ways of structuring Haskell programs by means of abstraction and introduce the technique of equational reasoning. For comprehensive accounts of the language including the discussion of features like lazy evaluation we refer the reader to recent textbooks [12,24].

2.1 A Short Tour of Haskell

Programming in Haskell means programming with equations. For example, a function delivering for every integer n, assumed to be nonnegative, the sum of the integer values between 0 and n is given as follows:

$$\begin{aligned} &\texttt{sum} :: \mathsf{Int} \to \mathsf{Int} \\ &\texttt{sum}\ 0 \qquad\ = 0 \\ &\texttt{sum}\ (n+1) = n + (\texttt{sum}\ n) \end{aligned}$$

Note the (optional) type signature, the use of recursion/induction, and the definition by cases. The above looks much like how a mathematician would typically write down a specification of the function `sum`, except for a different way of using parentheses in denoting function application.

Definition by cases is supported via so-called *pattern-matching* on the left-hand sides of equations, which is also available at other types than that of integers. For example, summing up the elements of a *list* of integer values can be done as follows:

$$\begin{aligned} &\texttt{listsum} :: [\mathsf{Int}] \to \mathsf{Int} \\ &\texttt{listsum}\ [\,] \qquad\ = 0 \\ &\texttt{listsum}\ (n : ns) = n + (\texttt{listsum}\ ns) \end{aligned}$$

The syntax of lists, as well as the way in which pattern-matching works for them, should become clear from the following example evaluation:

$$\begin{aligned}
&\texttt{listsum}\ [1,2,3,42] \\
&= 1 + (\texttt{listsum}\ [2,3,42]) \\
&= 1 + (2 + (\texttt{listsum}\ [3,42])) \\
&= 1 + (2 + (3 + (\texttt{listsum}\ [42]))) \\
&= 1 + (2 + (3 + (42 + (\texttt{listsum}\ [])))) \\
&= 1 + (2 + (3 + (42 + 0))) \\
&= 48
\end{aligned}$$

In addition to existing types like integers and lists, the user can define their own types at need, in particular arbitrary *algebraic data types*. For example, a type of binary, leaf-labelled integer trees is introduced as follows:

$$\textbf{data}\ \mathsf{Tree} = \mathsf{Node}\ \mathsf{Tree}\ \mathsf{Tree} \mid \mathsf{Leaf}\ \mathsf{Int}$$

Pattern-matching is automatically available for such user-defined types as well:

$$\begin{aligned}
&\texttt{treesum} :: \mathsf{Tree} \to \mathsf{Int} \\
&\texttt{treesum}\ (\mathsf{Leaf}\ n) \quad\ = n \\
&\texttt{treesum}\ (\mathsf{Node}\ t_1\ t_2) = (\texttt{treesum}\ t_1) + (\texttt{treesum}\ t_2)
\end{aligned}$$

Often, some functionality is useful at, and can indeed be uniformly specified for, more than one particular type. For example, computing the *length* of a list should be possible completely independently of the (type of) values contained *in* the list. The desired reuse here is enabled by *polymorphism*. A polymorphic type is one in which some concrete types are replaced by type variables. The length example then takes the following form:

$$\begin{aligned}
&\texttt{length} :: [\alpha] \to \mathsf{Int} \\
&\texttt{length}\ [\,] \quad\ = 0 \\
&\texttt{length}\ (a : as) = 1 + (\texttt{length}\ as)
\end{aligned}$$

This function can be used on lists of integers, [Int], lists of Boolean values, [Bool], and even lists of trees, [Tree], lists of lists of integers, [[Int]], and so on.

Polymorphism is not only available when defining functions, but also when defining types. For example, a more general version of the above tree data type, abstracted over the type of leaves, could have been introduced as follows:

$$\textbf{data}\ \mathsf{Tree}\ \alpha = \mathsf{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha) \mid \mathsf{Leaf}\ \alpha$$

Then we could still have `treesum` with exactly the same defining equations as above, but revised type signature $\texttt{treesum} :: \mathsf{Tree\ Int} \to \mathsf{Int}$, and moreover could write functions that do not depend on a particular type of leaf values. Like so, using the list concatenation operator $(+\!\!+) :: [\alpha] \to [\alpha] \to [\alpha]$:

$$\begin{aligned}
&\texttt{flatten} :: \mathsf{Tree}\ \alpha \to [\alpha] \\
&\texttt{flatten}\ (\mathsf{Leaf}\ a) \quad\ = [a] \\
&\texttt{flatten}\ (\mathsf{Node}\ t_1\ t_2) = (\texttt{flatten}\ t_1) +\!\!+ (\texttt{flatten}\ t_2)
\end{aligned}$$

List types are nothing special in Haskell. Except for some syntactic sugar, they are on an equal footing with user-defined algebraic data types. In fact, seeing [] as a type constructor of the same kind as the polymorphic version of Tree above, lists can be thought of as being introduced with the following definition:

$$\textbf{data}\ []\ \alpha = (:)\ \alpha\ ([]\ \alpha) \mid [\,]$$

Another important abstraction facility is the use of higher-order types. That is, a function argument can itself be a function. For example, the following function applies another function, which is supplied as an argument, to every element of an input list and builds an output list from the results:[1]

$$\begin{array}{l}\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\ \texttt{map}\ h\ [\,] \qquad = [\,] \\ \texttt{map}\ h\ (a : as) = (h\ a) : (\texttt{map}\ h\ as)\end{array}$$

Now *two* type variables, α and β, are used. They keep track of the dependencies between the argument and result types of h and the types of the input and output lists, respectively.

Since polymorphism, including forms of it that are more advanced than those already seen above, is at the heart of all results reported on here, the next subsection discusses it in some more detail.

2.2 The Virtues of Polymorphism

We have already introduced so-called *parametric polymorphism*, where the same algorithm is used for all different instantiations of a type variable. For the function $\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$ seen earlier this kind of polymorphism allows, for example:

$$\begin{array}{lll}\texttt{map}\ (+1)\ [1,2,3] & = [2,3,4] & \text{— with } \alpha, \beta \mapsto \mathsf{Int}, \mathsf{Int} \\ \texttt{map not}\ \ [\mathsf{True}, \mathsf{False}] & = [\mathsf{False}, \mathsf{True}] & \text{— with } \alpha, \beta \mapsto \mathsf{Bool}, \mathsf{Bool} \\ \texttt{map even}\ [1,2,3] & = [\mathsf{False}, \mathsf{True}, \mathsf{False}] & \text{— with } \alpha, \beta \mapsto \mathsf{Int}, \mathsf{Bool}\end{array}$$

The concrete choice of type parameters for α and β is not given explicitly in Haskell. Rather, it is inferred automatically (while, e.g., `map not` $[1,2,3]$ would be rejected).

So far, quantification over type variables has been implicit as well. For example, the type $(\alpha \to \beta) \to [\alpha] \to [\beta]$ is actually interpreted as $\forall\alpha.\forall\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$. The positioning and scope of quantifiers can be quite important. To see why, consider the following function definition:

$$\texttt{f}\ g = (g\ [1,2,3]) + (g\ [\mathsf{True}, \mathsf{False}])$$

[1] When reading such higher-order type signatures, the function arrow "→" associates to the right. So the type $(\alpha \to \beta) \to [\alpha] \to [\beta]$ is the same as $(\alpha \to \beta) \to ([\alpha] \to [\beta])$, but not the same as $\alpha \to \beta \to [\alpha] \to [\beta]$.

Note that in the equation's right-hand side the function g is applied to lists of two different types. But that should be fine if we apply `f`, for example, to `length` (in which case we would expect the overall result to be 5). So it might be tempting to give `f` the type $([\alpha] \to \mathsf{Int}) \to \mathsf{Int}$. But this attempt would fail, as it would mean that we expect `f` to be a polymorphic function which for any concrete type, say τ, takes a function of type $([\tau] \to \mathsf{Int})$ as argument and delivers an integer as result. And this τ is neither guaranteed to be Int, nor Bool, and certainly not both at the same time. So the function calls $g\ [1,2,3]$ and $g\ [\mathsf{True}, \mathsf{False}]$ are bound to lead to trouble. The point is that we do not really want `f` *itself* to be polymorphic, but rather want it to be a function that takes a polymorphic function as *argument*. That is, instead of $([\alpha] \to \mathsf{Int}) \to \mathsf{Int}$, which is equivalent to $\forall\alpha.([\alpha] \to \mathsf{Int}) \to \mathsf{Int}$, we need `f` to have the type $(\forall\alpha.[\alpha] \to \mathsf{Int}) \to \mathsf{Int}$. Such *rank-2 types* [15] are allowed in mature Haskell implementations, and are crucial for the mentioned program transformations [9,32], but also for the technique to be discussed in Section 5. It is important to note, though, that this additional abstraction facility, being able to write functions that abstract over functions that abstract over types, comes at the price of type signatures no longer being optional. In particular, the equation for `f` as given above in isolation is not a legal function definition. Only when we add the type signature[2]

$$\texttt{f} :: (\textbf{forall}\ \alpha.\ [\alpha] \to \mathsf{Int}) \to \mathsf{Int}$$

it is accepted by the type checker; and so is, then, `f length`, which computes 5.

Another form of polymorphism is the so-called *ad-hoc* one, where a certain functionality is provided for different types, without necessarily the same algorithm being used in each and every instantiation. For example, an equality test for lists of integers is likely to be implemented differently than the same kind of test for integers themselves, or for trees of Boolean values. In Haskell, such overloading of functionality is supported via *type classes* [36]. For example, the class Eq with methods == and /= is declared as follows:

$$\begin{aligned}&\textbf{class}\ \mathsf{Eq}\ \alpha\ \textbf{where}\\ &\quad (==) :: \alpha \to \alpha \to \mathsf{Bool}\\ &\quad (/=)\ \ :: \alpha \to \alpha \to \mathsf{Bool}\end{aligned}$$

For base types like Int these methods are predefined, while for other types they could be defined as in the following example:

$$\begin{aligned}&\textbf{instance}\ \mathsf{Eq}\ \alpha \Rightarrow \mathsf{Eq}\ [\alpha]\ \textbf{where}\\ &\quad [\,] \quad\ == [\,] \quad\ = \mathsf{True}\\ &\quad (x : xs) == (y : ys) = (x == y)\ \&\&\ (xs == ys)\\ &\quad xs \quad\ /= ys \quad\ = \texttt{not}\ (xs == ys)\end{aligned}$$

Here an equality test for elements of an arbitrary, but fixed, type is used for defining an equality test for lists of elements of that type. Without further definitions, the methods == and /= are then available for [Int], [[Int]], and so on.

[2] Compiler flag `-XRank2Types` of GHC (`http://www.haskell.org/ghc`) version 6.8.2 is used from now on.

And the same is true for functions defined in terms of them, such as the following one:

$$\begin{aligned}&\texttt{elem} :: \textbf{forall}\ \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow \alpha \to [\alpha] \to \mathsf{Bool}\\ &\texttt{elem}\ x = \texttt{foldr}\ (\lambda a\ r \to (a == x)\ ||\ r)\ \mathsf{False}\end{aligned}$$

2.3 Equational Reasoning

As seen, a Haskell program is just a collection of equations defining the return values of functions for given arguments. This approach is fundamentally different from the concept of functions or procedures in imperative or impure functional languages, where they may additionally access, and alter, some global state. A Haskell function is really a function in the mathematical sense, transferring values to values and doing nothing else. This absence of side-effects implies that every expression has a value that is independent of when it is evaluated. Clearly, two expressions having the same value can thus be replaced for each other in any program context without changing the overall semantics; a property often called *referential transparency*. And the easiest way to establish that two expressions have the same value is to observe them as the left- and right-hand sides of the same program equation. Of course, doing so might involve the instantiation of variables, on both sides and in exactly the same manner, that stand for abstracted parts of the function's input. Overall, this approach leads to a simple but powerful reasoning principle.

Since the above explanation is best substantiated by an example, we consider the following function definition:

$$\begin{aligned}&\texttt{filter} :: \textbf{forall}\ \alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]\\ &\texttt{filter}\ p\ [\,] \qquad = [\,]\\ &\texttt{filter}\ p\ (a : as) = \textbf{if}\ p\ a\ \textbf{then}\ a : (\texttt{filter}\ p\ as)\\ &\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \texttt{filter}\ p\ as\end{aligned}$$

Assume we want to prove that for every choice of p, h, and as (of appropriate types), the following law holds:

$$\texttt{filter}\ p\ (\texttt{map}\ h\ as) = \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ as) \tag{1}$$

Proceeding by induction on the list as, it suffices to establish that

$$\texttt{filter}\ p\ (\texttt{map}\ h\ [\,]) = \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ [\,]) \tag{2}$$

holds and that

$$\texttt{filter}\ p\ (\texttt{map}\ h\ (a : as)) = \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ (a : as)) \tag{3}$$

holds under the assumption that the induction hypothesis (1) holds for as. For the induction base (2), equational reasoning succeeds as follows:

$$\begin{aligned}&\quad\texttt{filter}\ p\ (\texttt{map}\ h\ [\,])\\ &= \texttt{filter}\ p\ [\,]\\ &= [\,]\\ &= \texttt{map}\ h\ [\,]\\ &= \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ [\,])\end{aligned}$$

And for the induction step (1)→(3):

$$
\begin{aligned}
& \texttt{filter}\ p\ (\texttt{map}\ h\ (a : as)) \\
= {} & \texttt{filter}\ p\ ((h\ a) : (\texttt{map}\ h\ as)) \\
= {} & \textbf{if}\ p\ (h\ a)\ \textbf{then}\ (h\ a) : (\texttt{filter}\ p\ (\texttt{map}\ h\ as)) \\
& \qquad \textbf{else}\ \ \texttt{filter}\ p\ (\texttt{map}\ h\ as) \\
= {} & \textbf{if}\ (p \circ h)\ a\ \textbf{then}\ (h\ a) : (\texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ as)) \\
& \qquad \textbf{else}\ \ \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ as) \\
= {} & \textbf{if}\ (p \circ h)\ a\ \textbf{then}\ \texttt{map}\ h\ (a : (\texttt{filter}\ (p \circ h)\ as)) \\
& \qquad \textbf{else}\ \ \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ as) \\
= {} & \texttt{map}\ h\ (\textbf{if}\ (p \circ h)\ a\ \textbf{then}\ a : (\texttt{filter}\ (p \circ h)\ as) \\
& \qquad\qquad\qquad \textbf{else}\ \ \texttt{filter}\ (p \circ h)\ as) \\
= {} & \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ (a : as))
\end{aligned}
$$

While equational reasoning is employed as an auxiliary technique in the works we report on, our main focus is on reasoning about functions *without* having access to their defining equations. How such reasoning is possible is the subject of the next section.

3 Free Theorems

We review why a polymorphic type may allow to derive statements about a function's behaviour without knowing that function's defining equations.

3.1 Free Theorems, Intuitively

It is best to start with a concrete example. Consider the following type signature:

$$\texttt{f} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]$$

What does it tell us about the function `f`? For sure that it takes lists as input and produces lists as output. But we also see that `f` is polymorphic, and so must work for lists over arbitrary element types. How, then, can elements for the output list come into existence? The answer is that the output list can only ever contain elements from the input list. This is so because the function `f`, not knowing the element type of the lists it operates over, cannot possibly make up new elements of any concrete type to put into the output, such as 42 or `True`, or even `id`, because then `f` would immediately fail to have the general type **forall** $\alpha.\ [\alpha] \to [\alpha]$.

> So for any input list l (over any element type) the output list `f` l consists solely of elements from l.

But how can `f` decide which elements from l to propagate to the output list, and in which order and multiplicity? The answer is that such decisions can only be made based on the input list l. For `f` has no access to any global state or other context based on which to decide. It cannot, for example, consult the user in

any way about what to do. The means by which to make decisions based on l are quite limited as well. In particular, decisions cannot possibly depend on any specifics of the elements of l. For the function `f` is ignorant of the element type, and so is prevented from analysing list elements in any way (be it by pattern-matching, comparison operations, or whatever). In fact, the only means for `f` to drive its decision-making is to inspect the *length* of l, because that is the only element-independent "information content" of a list.

> So for any pair of lists l and l' of same length (but possibly over different element types) the lists `f` l and `f` l' are formed by making the same position-wise selections of elements from l and l', respectively.

Now recall the function `map` from Section 2.1. Clearly, `map` h for any function h preserves the lengths of lists. So if $l' =$ `map` $h\ l$, then `f` l and `f` l' are of the same length and contain, at each position, position-wise exactly corresponding elements from l and l', respectively. Since, moreover, any two position-wise corresponding elements, one from l and one from $l' =$ `map` $h\ l$, are related by the latter being the h-image of the former, we have that at each position `f` l' contains the h-image of the element at the same position in `f` l.

> So for any list l and (type-appropriate) function h, the following law holds:
>
> $$\texttt{f}\ (\texttt{map}\ h\ l) = \texttt{map}\ h\ (\texttt{f}\ l) \tag{4}$$

Note that during the reasoning leading up to this statement we did not (need to) consider the actual definition of `f` at all. It could have been `f = reverse`, or `f = tail`, or `f = take` 5, or many other choices. It just did not matter.

And what we just noticed is not a one-off success. Intuitive reasoning of the same style as above can be applied to other polymorphic functions as well. For example, one can arrive at the conclusion that for every function

$$\texttt{f} :: \textbf{forall}\ \alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

the following law holds:

$$\texttt{f}\ p\ (\texttt{map}\ h\ l) = \texttt{map}\ h\ (\texttt{f}\ (p \circ h)\ l) \tag{5}$$

The steps required to establish this law are but minor extensions of the ones leading to law (4) above. It is only necessary to additionally factor in how `f`'s decision about which elements from an input list to propagate to the output list, and in which order and multiplicity, may now depend also on the outcomes of an input predicate, namely `f`'s first argument, on the input list's elements.

Note that law (5) is exactly the same as law (1) in Section 2.3, except that now we claim it much more generally for all functions of `filter`'s type, not just for the particular one considered there. And there is no need for induction anymore. Better yet, the intuitive reasoning above can be put on a more formal basis. The methodology of deriving free theorems à la Wadler [34] provides precisely a way to obtain statements like above for arbitrary function types, and in a more disciplined (and provably sound) manner than mere handwaving.

3.2 The Formal Background of Free Theorems

The origin of free theorems lies in Reynolds' [26] studies about characterising parametric polymorphism. The question approached was what it means for a polymorphic function to behave uniformly, regardless of the concrete type at which it is instantiated. Intuitively, the concept of two functions to behave the same is that they map equal arguments to equal results. But this intuition does not really make sense when we want to compare two different instantiations of a polymorphic function. For example, when trying to compare the two instantiations of a function `f` :: **forall** α. $[\alpha] \to [\alpha]$ at types Int and Bool, we cannot say that they "behave the same" when they "map equal input lists to equal output lists". After all, one of the two instantiations under consideration maps integer lists to integer lists, while the other maps Boolean valued lists to Boolean valued lists. And there is no concept of an integer list being "equal" to a Boolean valued list. Reynolds' key idea was to move away from equality and instead consider arbitrary binary relations. Say we fix a relation between Int and Bool that relates every even integer to True and every odd integer to False. Given this relation as a base, it is straightforward to formulate a meaningful concept of an integer list and a Boolean valued list being related: we simply require that the lists are of the same length and that elements at corresponding positions are related in the way just described. For the supposed two instances of `f` at Int and Bool we can now require that they map *related* input lists to *related* output lists. If `f` is truly polymorphic, with its behaviour independent of concrete choices for instantiating α, then this invariant will indeed be preserved. Actually, it will be so for *every* choice of a base relation between Int and Bool, not just for the one connecting even integers to True and odd ones to False. And what is more, this condition is not only necessary, but also sufficient. Not only will every truly polymorphic `f` preserve every relation between every pair of concrete types chosen for instantiating α, but also conversely is this universal preservation enough to establish that `f` is polymorphic in a truly uniform way. For any `f` that were to "cheat" by behaving differently for one type or another, it would be possible to find some relation that is not preserved. It is this characterisation on which Wadler [34] built his methodology of deriving free theorems. Of course, it is necessary to make precise the idea of propagating relations from the base level to relations over lists, over functions, and so on.

First, every quantification over type variables is replaced by quantification over relation variables. For example, given the type signature `f` :: **forall** α. $[\alpha] \to [\alpha]$ we obtain $\forall\mathcal{R}$. $[\mathcal{R}] \to [\mathcal{R}]$. Then, there is a systematic way of reading such expressions over relations as relations themselves. In particular,

- base types like Int are read as identity relations,
- for relations $\mathcal{R}$ and $\mathcal{S}$ we have

$$\mathcal{R} \to \mathcal{S} = \{(f, g) \mid \forall (a, b) \in \mathcal{R}.\ (f\ a, g\ b) \in \mathcal{S}\}$$

and

– for "type schemes" τ and τ' with at most one free variable, say α, and a function $\mathcal{F}$ on relations such that every relation $\mathcal{R}$ between concrete types τ_1 and τ_2, denoted $\mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2)$, is mapped to a relation $\mathcal{F}\ \mathcal{R} \in \mathit{Rel}(\tau[\tau_1/\alpha], \tau'[\tau_2/\alpha])$, we have

$$\forall \mathcal{R}.\ \mathcal{F}\ \mathcal{R} = \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2).\ (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}\ \mathcal{R}\}$$

(Here, u_{τ_1} is the instantiation of a value u of type **forall** $\alpha.\ \tau$ at the type τ_1, and similarly for v_{τ_2}. So far, we have always left type instantiation implicit, and we will continue to do so in what follows.)

Also, every type constructor is read as an appropriate construction on relations. For example, the list type constructor maps every relation $\mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2)$ to the relation $[\mathcal{R}] \in \mathit{Rel}([\tau_1], [\tau_2])$ defined by

$$[\mathcal{R}] = \{([\,], [\,])\} \cup \{(a : as, b : bs) \mid (a, b) \in \mathcal{R},\ (as, bs) \in [\mathcal{R}]\}$$

and similarly for other algebraic data types.

Free theorems are now derived from the fact, proved once and then used over and over again, that every value of a concrete type is related to itself by the relational interpretation of that type. For the example `f` :: **forall** $\alpha.\ [\alpha] \to [\alpha]$ this fact implies that any such `f` satisfies $(\texttt{f}, \texttt{f}) \in \forall \mathcal{R}.\ [\mathcal{R}] \to [\mathcal{R}]$, which by unfolding some of the above definitions is equivalent to having for every τ_1, τ_2, $\mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2)$, $l :: [\tau_1]$, and $l' :: [\tau_2]$ that $(l, l') \in [\mathcal{R}]$ implies $(\texttt{f}\ l, \texttt{f}\ l') \in [\mathcal{R}]$, or, specialised to the function level ($\mathcal{R} \mapsto h$, and thus $[\mathcal{R}] \mapsto \texttt{map}\ h$), for every $h :: \tau_1 \to \tau_2$ and $l :: [\tau_1]$ that $\texttt{f}\ (\texttt{map}\ h\ l) = \texttt{map}\ h\ (\texttt{f}\ l)$. This calculation finally provides the formal, and systematic, counterpart to the intuitive reasoning seen earlier.

4 A Knuth-like 0-1-2-Principle for Parallel Prefix Computation

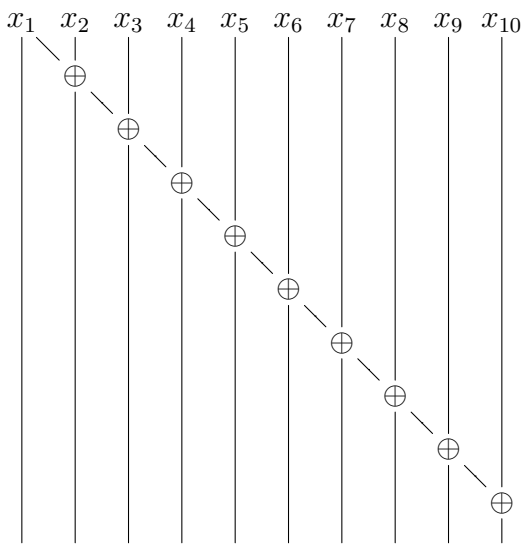
This section presents an application of type-based reasoning to a real-world problem [31]. In particular, we benefit from Haskell's mathematical rigour and its abstraction and reasoning facilities in the endeavour to analyse a whole class of algorithms.

4.1 Parallel Prefix Computation

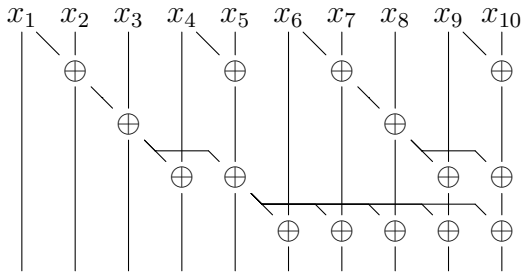
Parallel prefix computation is a task with numerous applications in the hardware and algorithmics fields [3]. The basic problem description is as follows:

> Given an associative binary operation $\oplus$ and inputs $x_1, \ldots, x_n$, compute the values x_1, $x_1 \oplus x_2$, $x_1 \oplus x_2 \oplus x_3$, ... up to $x_1 \oplus x_2 \oplus \cdots \oplus x_n$.

Here is an obvious solution for $n = 10$, depicted as a *prefix network* in which the inputs are provided at the top, values flow downwards along "wires" and get combined by $\oplus$-"gates", and the outputs can be read off at the bottom, from left to right:



At first glance, the above may appear to be the best solution possible, as it employs maximal reuse of partial results. After all, it is clear that nine applications of $\oplus$ are necessary to compute $x_1 \oplus x_2 \oplus \cdots \oplus x_{10}$ alone. So if the same nine applications yield all the other required outputs as well, what could be better? The point is that the number of applications of $\oplus$ is not the only measure of interest. For example, the above solution is inherently sequential, which leads to bad time performance even on parallel devices. Assuming that each application of $\oplus$ requires one unit of time, the last output is not available until nine units have passed. In contrast, the following maximally parallel solution requires only four time units to deliver all outputs:

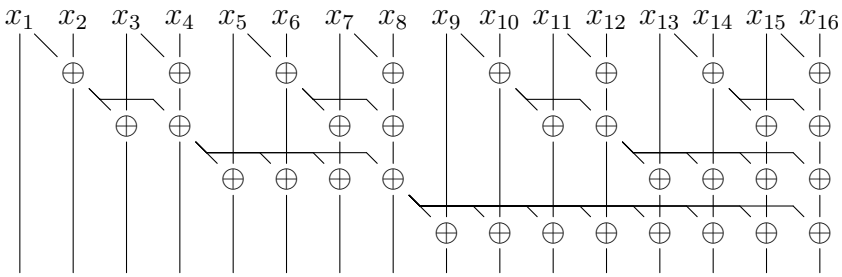


Note that thanks to the assumed associativity of $\oplus$, correctness is still guaranteed. For example, $x_1 \oplus x_2 \oplus \cdots \oplus x_7$ is now actually computed as $(((x_1 \oplus x_2) \oplus x_3) \oplus (x_4 \oplus x_5)) \oplus (x_6 \oplus x_7)$.

Admittedly, the shorter time to output in the parallel solution comes at the expense of an increased number of $\oplus$-"gates" and more complicated "wiring". But depending on the usage scenario this increase can be a worthwhile allowance. In some scenarios, in particular in a hardware setting where the "wires" are

real wires and the $\oplus$-"gates" are real gates, many more trade-offs (guided by architectural, delay, or other constraints) are possible and of potential benefit. Hence, a wealth of solutions has been developed over the years [28,5,16]. Key to all of them is to use the associativity of $\oplus$ to rearrange how partial results are computed and combined.

An obvious concern is that for correctness of such new, and increasingly complex, methods. While checking the correctness of a concrete prefix network is a straightforward, though maybe tedious, task, the real practical interest is in validating a whole method of constructing prefix networks. For that is the general nature of work on parallel prefix computation: to develop and study algorithms that yield networks for arbitrary $n \geq 1$. In the case of the completely sequential network it should be clear how to abstract from $n = 10$ to arbitrary n. But also behind the other network shown above there is a general construction principle. It is the method of Sklansky [28], and as another example here is its instance for $n = 16$:



So if studying prefix networks really means studying methods for their construction, how should these be expressed? Clearly, it would be beneficial to have a common framework in which to describe all methods, be they classical or still under development. For then they could be more readily compared, maybe combined, and hopefully analysed using a common set of reasoning principles, as opposed to when each method is described in a different formalism or notation. One very attractive choice for the unifying framework is to use some universal programming language. After all, by Turing completeness, this choice would allow to precisely capture the notion of an *algorithm* that may, or may not, be a correct solution to the parallel prefix computation task. Of course, this idea begs the question in terms of *which* programming language to cast the problem, algorithms, and analyses. It turns out that Haskell, with its mathematical expressivity and nice abstraction facilities, is a very good fit.

4.2 Prefix Networks in Haskell

From the problem description at the beginning of the previous subsection it is clear that any function implementing parallel prefix computation should have the following type:

$$\textbf{forall}\ \alpha.\ (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

The polymorphism over α is justified by the fact that in the problem description neither the type of the inputs $x_1, \ldots, x_n$, nor any specifics (apart from associativity) of $\oplus$ are fixed, except (implicitly) that the type on which $\oplus$ operates should be the same as that of the inputs. By providing the inputs in a variable-length list, we express our interest in algorithms that work for arbitrary n. And indeed, the prefix networks seen in the previous subsection are easily generated in Haskell.

For example, the completely sequential construction method is captured as follows:

$$\begin{aligned}
&\texttt{serial} :: \textbf{forall}\ \alpha.\ (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\texttt{serial}\ op\ (x : xs) = go\ x\ xs \\
&\quad \textbf{where}\ go\ x\ [\,] \quad\ = [x] \\
&\qquad\qquad go\ x\ (y : ys) = x : (go\ (x\ \text{`}op\text{`}\ y)\ ys)
\end{aligned}$$

The method of Sklansky [28] is captured as follows:

$$\begin{aligned}
&\texttt{sklansky} :: \textbf{forall}\ \alpha.\ (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\texttt{sklansky}\ op\ [x] = [x] \\
&\texttt{sklansky}\ op\ xs\ = us +\!\!+ vs \\
&\quad \textbf{where}\ t \quad\ = ((\texttt{length}\ xs) + 1)\ \text{`}\texttt{div}\text{`}\ 2 \\
&\qquad\qquad (ys, zs) = \texttt{splitAt}\ t\ xs \\
&\qquad\qquad us \quad\ = \texttt{sklansky}\ op\ ys \\
&\qquad\qquad u \quad\ = \texttt{last}\ us \\
&\qquad\qquad vs \quad\ = \texttt{map}\ (u\ \text{`}op\text{`})\ (\texttt{sklansky}\ op\ zs)
\end{aligned}$$

This function is already a bit more complicated than `serial`, but still expressed in a way that is nicely declarative and accessible. Confidence that this code really implements Sklansky's method can also be gained from the fact that the two parallel network pictures shown in the previous subsection, for $n = 10$ and $n = 16$, were automatically generated from it.[3] And more recent algorithms for parallel prefix computation can be treated in the same way.

What is only hinted at above actually extends to a whole methodology for designing, and then analysing, hardware circuits using functional languages. An interesting introductory text containing many references is [27]. Our contribution reported here is a display of what powerful abstractions can buy in this context. Its reasoning is specific to parallel prefix computation, but similar results may hold for other algorithm classes of interest.

4.3 A Knuth-like 0-1-2-Principle

Assume we have a candidate function of type **forall** $\alpha.\ (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$. This function could be our attempt at implementing a classical method from the literature. Or it could be a new algorithm we have come up with or obtained by

[3] In the same way, namely using a separate Haskell program, the completely sequential network picture for $n = 10$ was automatically generated from the function `serial` above.

refining or combining existing ones. Indeed, it could be a function that we only hope to correctly implement parallel prefix computation, while actually it does not. To assure ourselves of its correctness, we may try an explicit proof or at least perform systematic testing. But it seems that in order to do so, we would have to consider every concrete type τ as potential instantiation for α, and for each such τ consider every (associative) operation of type $\tau \to \tau \to \tau$ as well as every input list of type $[\tau]$. Not only would this mean a lot of work, it is also unsatisfactory on a conceptual level. After all, given the rather generic problem description, we could expect that analyses of solution candidates are possible in a sufficiently generic way as well.

Here the *0-1-Principle* of Knuth [14] comes to mind. It states that if an oblivious sorting algorithm, that is one where the sequence of comparisons performed is the same for all input sequences of any given length, is correct on Boolean valued input sequences, then it is correct on input sequences over any totally ordered value set. This principle greatly eases the analysis of such algorithms. Is something similar possible for parallel prefix computation? For 0-1 the answer is negative: one can give a function that is correct for all binary operations and input lists over Boolean values, but not in general. The next best thing to hope for then is that a three-valued type may suffice as a discriminator between good and bad candidate functions. And this is indeed the case.

Our 0-1-2-Principle for parallel prefix computation can be formulated as follows. Let a function

$$\texttt{candidate} :: \textbf{forall}\ \alpha.\ (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

be given and let

$$\textbf{data}\ \mathsf{Three} = \mathsf{Zero} \mid \mathsf{One} \mid \mathsf{Two}$$

If for every associative operation $(\oplus) :: \mathsf{Three} \to \mathsf{Three} \to \mathsf{Three}$ and every list $xs :: [\mathsf{Three}]$,

$$\texttt{candidate}\ (\oplus)\ xs = \texttt{serial}\ (\oplus)\ xs$$

then the same holds for every type τ, associative $(\oplus) :: \tau \to \tau \to \tau$, and $xs :: [\tau]$. That is, correctness of `candidate` at the type Three implies its correctness at arbitrary type. Here the definition of "correctness" is "semantic equivalence to `serial` for associative operations as first input". Actually, the formal account [31] uses a different reference implementation than `serial`, but one that is easily shown to be semantically equivalent to it by equational reasoning.

The only aspect of the overall proof to which we want to draw attention here is the role of type-based reasoning. Note that we have not put any restriction on the actual definition of `candidate`, just on its type. This situation is, of course, a case for working with a free theorem. The free theorem derived from `candidate`'s type is that for every choice of concrete types τ_1 and τ_2, a function $h :: \tau_1 \to \tau_2$, and operations $(\otimes) :: \tau_1 \to \tau_1 \to \tau_1$ and $(\oplus) :: \tau_2 \to \tau_2 \to \tau_2$, if for every $x, y :: \tau_1$,

$$h\ (x \otimes y) = (h\ x) \oplus (h\ y) \tag{6}$$

then for every $xs :: [\tau_1]$,

$$\texttt{map}\ h\ (\texttt{candidate}\ (\otimes)\ xs) = \texttt{candidate}\ (\oplus)\ (\texttt{map}\ h\ xs)$$

This free theorem's conclusion gives us a starting point for relating the behaviour of `candidate` at different types, as ultimately required for the 0-1-2-Principle. Unfortunately, it is not as easy as setting $\tau_1 = \mathsf{Three}$ and $\tau_2 = \tau$ and working from there. Instead, we found it necessary to use an indirection via the type of integer lists (and an auxiliary statement originally discovered by M. Sheeran). Also, some good choices for h, $\otimes$, and $\oplus$ must be made, associativity must be factored into establishing the precondition (6), and some properties of permutations are needed. But all in all, once we have the above free theorem, the proof is mainly a bunch of equational reasoning steps. It has additionally been machine-verified using the Isabelle interactive proof assistant [4].

5 Semantic Bidirectionalisation

This section presents a novel approach to the view-update problem known from the database area, utilising programming language theory surrounding polymorphic types [33].

5.1 Bidirectional Transformation

Assume we have a domain of concrete values and a function `get` that takes such a value as source and produces from it a view by abstracting from some details. Now assume this view is updated in some way, and we would like to propagate this change back to the input source. So we need another function `put` that takes the original source and an updated view and produces an updated source. Clearly, `get` and `put` should be suitably related, because otherwise the integrity of the data to be transformed by using them is threatened. In the database area, where the concrete and abstract domains will typically be relation tables or XML trees, the following conditions have been proposed [1]:

$$\texttt{put}\ s\ (\texttt{get}\ s) = s \tag{7}$$

$$\texttt{get}\ (\texttt{put}\ s\ v) = v \tag{8}$$

$$\texttt{put}\ (\texttt{put}\ s\ v)\ (\texttt{get}\ s) = s \tag{9}$$

$$\texttt{put}\ (\texttt{put}\ s\ v)\ v' = \texttt{put}\ s\ v' \tag{10}$$

known as acceptability, consistency, undoability, and composability.

Writing and maintaining good `get`/`put`-pairs requires considerable effort. So it is natural to invest in methodologies that can reduce this burden on the programmer. The ideal is to not have to write two separate specifications and to establish their relatedness by proving (some of) the conditions above, but to instead be able to provide only a single specification and still get both forward/backward-components. This problem has received much attention from

the programming language community in recent years. For example, Foster et al. [8] pioneered a domain-specific language approach that fences in a certain subclass of transformations, provides a supply of correctly behaving `get`/`put`-pairs on a low level, and then describes systematic and sound ways of assembling bigger bidirectional transformations from smaller ones. Another approach is to devise an algorithm that works on a syntactic representation of somehow restricted `get`-functions and tries to infer appropriate `put`-functions automatically [17]. While all the approaches proposed in the literature had been syntactic in nature, we present one that works purely on the level of semantic values.

5.2 Bidirectionalisation of Polymorphic `get`

The idea is to write, directly in the language in which the forward and backward functions shall live themselves, a higher-order function that takes `get`-functions as arguments and returns appropriate `put`-functions. It turns out that Haskell is very well up to the task. One thing to stress is that "on the semantic level" means that when prescribing how `put` will behave we are not willing, or even able, to inspect the function definition of `get`. That is, the backward component we return cannot be based on a deep analysis of the forward function's innards. This restriction may sound crippling, and yet we can provide nontrivial, and well-behaved, `put`-functions for a wide range of (polymorphic) `get`-functions. And forgoing any possibility to "look into" `get` liberates our approach from considerable syntactic restraints. In particular, and in contrast to the situation with all previous approaches, the programmer is not anymore restricted to drawing forward functions from some sublanguage only.

Let us consider a specific example in Haskell, for simplicity working with lists only rather than with richer data structures like tables or trees. Assume our `get`-function is as follows:

```
get :: forall α. [α] → [α]
get as = take ((length as) `div` 2) as
```

Here the abstraction amounts to omitting the input list's second half. Propagating an update on the preserved first half back to the original, full list can be done with the following function:

```
put :: forall α. [α] → [α] → [α]
put as as' = let n = (length as) `div` 2
             in  if (length as') == n then as' ++ (drop n as)
                                      else  error "Shape mismatch."
```

And indeed, our higher-order function `bff` (named for an abbreviation of the full paper's title), when applied to the above `get`, will return this `put`.[4] Of course not the exact syntactic definition of `put` that is shown above, but a functional value

[4] Well, almost. Actually, it will return this function with type **forall** α. `Eq` $\alpha \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$.

that is semantically equivalent to it. This is absolutely enough from an application perspective. We want automatic bidirectionalisation precisely because we do not want to be bothered with thinking about the backward function. So we do not care about its syntactic form, as long as the function serves its purpose. There is a certain price to pay, namely `bff get` runs much less efficiently on its inputs than the hand-coded `put` does, in this and in other examples. But that is a different story. Here we are interested in safety and programmer (rather than program) productivity.

One aspect to be aware of is that the `put`-function given above is a partial function only. That is, it may raise an exception for unsuitable input that represents a view-update that cannot (automatically and consistently) be reconciled with the original source. Some in the related literature, notably [8] and follow-on works, emphasise the static description, or even calculation, of the domain on which a `put`-function is totally defined. We instead follow Matsuda et al. [17], accept partiality, and weaken the bidirectional properties (8)–(10) somewhat by adding definedness preconditions. Specifically, these three properties are only required to hold if `put` s v is actually defined, and (10) even has the additional precondition that `put` (`put` s v) v' is defined as well. The thus revised conditions, and the original (7), are what we prove for polymorphic `get` and `put` = `bff get`.[5] The way we do so crucially depends on `get` being of polymorphic type, because such a type allows us to learn something about `get`'s behaviour without having access to its defining equations.

5.3 Leveraging Free Theorems

We do not want to repeat the full development and implementation of `bff` or the associated proofs here, but at least explain some of the key ideas.

Assume that `bff` is given a function `get` :: **forall** α. $[\alpha] \to [\alpha]$ as input. How can it gain information about this function, so as to exploit that information for producing a good backward function? Note that `get` is of exactly the type discussed as first example in Section 3.1. There, we have analysed what this type tells us about the behaviour of any such function. The essence of this analysis was that such a function's behaviour does not depend on any concrete list elements, but only on positional information. Now we additionally use that this positional information can even be observed explicitly, for example by applying `get` to ascending lists over integer values. Say `get` is `tail`, then every list $[0..n]$ is mapped to $[1..n]$, which allows `bff` to see that the head element of the original source is absent from the view, hence cannot be affected by an update on the view, and hence should remain unchanged when propagating an updated view back into the source. And this observation can be transferred to other source lists than $[0..n]$ just as well, even to lists over non-integer types, thanks to law (4) from Section 3.1. In particular, that law allows us to establish that for every list

[5] Again, almost. In general, we prove the conditions up to == rather than up to semantic equivalence. But for the typical instances of `Eq` used in practice, == and = totally agree.

s of the same length as $[0..n]$, but over arbitrary type, we have

$$\texttt{get}\ s = \texttt{map}\ (s\,!!)\ (\texttt{get}\ [0..n]) \tag{11}$$

where (!!) :: **forall** α. $[\alpha] \to \mathsf{Int} \to \alpha$ is the operator used in Haskell for extracting a list element at a given index position, starting counting from 0.

Let us develop the above line of reasoning further, again on the `tail` example. So `bff tail` is supposed to return a good `put`. To do so, it must determine what this `put` should do when given an original source s and an updated view v. First, it would be good to find out to what element in s each element in v corresponds. Assume s has length $n+1$. Then by applying `tail` to the same-length list $[0..n]$, `bff` (or, rather, `bff tail` = `put`) learns that the original view from which v was obtained by updating had length n, and also to what element in s each element in that original view corresponded. Being conservative, we will only accept v if it has retained that length n. For then, we also know directly the associations between elements in v and positions in the original source. Now, to produce the updated source, we can go over all positions in $[0..n]$ and fill them with the associated values from v. For positions for which there is no corresponding value in v, because these positions were omitted when applying `tail` to $[0..n]$, we can look up the correct value in s rather than in v. For the `tail` example, this will only concern position 0, for which we naturally take over the head element from s.

The same strategy works also for general `bff get`. In short, given s, produce a kind of template $s' = [0..n]$ of the same length, together with an association g between integer values in that template and the corresponding values in s. Then apply `get` to s' and produce a further association h by matching this template view versus the updated proper value view v. Combine the two associations into a single one h', giving precedence to h whenever an integer template index is found in both h and g. Thus, it is guaranteed that we will only resort to values from the original source s when the corresponding position did not make it into the view, and thus there is no way how it could have been affected by the update. Finally, produce an updated source by filling all positions in $[0..n]$ with their associated values according to h'. Some extra care is needed when matching the template view versus the updated proper value view, to produce h, for the case that an index position is encountered twice. This case occurs as soon as `get` duplicates a list element. Consider, for example, $\texttt{get} = (\lambda s \to s +\!\!+ s)$. Applied to a template $[0..n]$, it will deliver the template view $[0, \ldots, n, 0, \ldots, n]$. Under what conditions should a match between this template view and an updated proper value view be considered successful? Clearly only when equal indices match up with equal values, because only then we can produce a meaningful association reflecting a legal update.

Using the standard functions

$$\texttt{zip} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ [\alpha] \to [\beta] \to [(\alpha, \beta)]$$

and

$$\texttt{lookup} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ \mathsf{Eq}\ \alpha \Rightarrow \alpha \to [(\alpha, \beta)] \to \mathsf{Maybe}\ \beta$$

with

$$\textbf{data}\ \mathsf{Maybe}\ \beta = \mathsf{Nothing} \mid \mathsf{Just}\ \beta$$

and the obvious semantics, the strategy described above could be implemented as follows:

```
bff :: (forall α. [α] → [α]) → (forall α. Eq α ⇒ [α] → [α] → [α])
bff get [] [] = []
bff get [] v = error "Shape mismatch."
bff get s v = let s' = [0..((length s) − 1)]
                  g  = zip s' s
                  h  = assoc (get s') v
                  h' = h ++ g
              in  map (λi → case lookup i h' of Just b → b) s'

assoc :: Eq α ⇒ [Int] → [α] → [(Int, α)]
assoc []       []       = []
assoc (i : is) (b : bs) = let m = assoc is bs
                          in  case lookup i m of
                                Nothing → (i, b) : m
                                Just c  → if b == c
                                          then m
                                          else error "Update violates equality."
assoc _        _        = error "Shape mismatch."
```

Note that the first two defining equations for `bff` reflect the fact that a function `get` :: **forall** α. $[\alpha] \to [\alpha]$ can map $[\,]$ only to $[\,]$, so only an empty list is accepted as updated view for an empty source. The **case** and **if** in the second defining equation for `assoc` provide for the correct treatment of duplication of list elements, by checking whether indeed equal indices match up with equal values.

The implementation given above is clearly not optimal. It makes a rather bad choice for representing the associations between integer values and values from s and v. Above, lists of pairs are used for this, namely $[(\mathsf{Int}, \alpha)]$, and `lookup` is just linear search. The full paper [33] actually uses the standard library Data.IntMap instead, with better asymptotic behaviour. The implementation in the paper also differs in other, smaller ways from the one above, such as by a more refined error handling, but the key ideas are the same.

More importantly, the paper then goes on to develop semantic bidirectionalisation for other functions than ones of type **forall** α. $[\alpha] \to [\alpha]$. One dimension of generalisation is to consider functions that are not fully polymorphic, but may actually perform *some* operations on list elements. For example, the following function uses equality, or rather inequality, tests to remove duplicate occurrences of list elements:

```
get :: forall α. Eq α ⇒ [α] → [α]
get []       = []
get (a : as) = a : (get (filter (a /=) as))
```

It is not in the reach of the bidirectionalisation strategy described thus far. It cannot be given the type **forall** α. $[\alpha] \to [\alpha]$, and indeed the essential law (11) does not hold for it.[6] But by working with refined free theorems [34, Section 3.4] it is possible to treat `get`-functions of type **forall** α. Eq $\alpha \Rightarrow [\alpha] \to [\alpha]$ as well, to implement a higher-order function

$$\texttt{bff}_{\mathsf{Eq}} :: (\textbf{forall}\ \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \to (\textbf{forall}\ \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])$$

and to prove that every pair `get` :: **forall** α. Eq $\alpha \Rightarrow [\alpha] \to [\alpha]$ and `put` = $\texttt{bff}_{\mathsf{Eq}}$ `get` satisfies the laws (7)–(10), in their revised form discussed at the end of Section 5.2. The same goes for the type class Ord capturing ordering tests, a new higher-order function

$$\texttt{bff}_{\mathsf{Ord}} :: (\textbf{forall}\ \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \to (\textbf{forall}\ \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])$$

and `get`-functions like the following one:

$$\begin{aligned} &\texttt{get} :: \textbf{forall}\ \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \\ &\texttt{get} = (\texttt{take}\ 3) \circ \mathsf{List}.\texttt{sort} \end{aligned}$$

For each of `bff`, $\texttt{bff}_{\mathsf{Eq}}$, and $\texttt{bff}_{\mathsf{Ord}}$, the full paper actually only discusses the proofs for conditions (7) and (8), but those for (9) and (10) are similar.

Another dimension of generalisation is to consider functions that deal with data structures other than lists. By employing polymorphism over *type constructor classes* [13], like Haskell's Functor class, and type-generic programming techniques, we provide one implementation of each `bff`, $\texttt{bff}_{\mathsf{Eq}}$, and $\texttt{bff}_{\mathsf{Ord}}$ that applies to functions involving a wide range of type constructors, on both the source and the view sides. For example, the very same `bff` can be used to bidirectionalise the `get`-function shown in Section 5.2 as well as `flatten` from Section 2.1.

An online interface to the implementations from the full paper is accessible at `http://linux.tcs.inf.tu-dresden.de/~bff/cgi-bin/bff.cgi`.

6 Outlook

The story of type-based thinking and reasoning about programs is only at its beginning. We expect to see it having a big impact also on practical software construction in the coming years. One possible scenario is that ideas and features first developed and studied in the context of perceivedly mere academic languages and type systems continue to slowly trickle into the mainstream, as has happened with the inclusion of parametric polymorphism into Java and C#, and of first-class functions (i.e., λ-abstractions) into the latter. Scala [23] is an experiment into how a resulting hybrid object-oriented/functional language with strong ties to type theory research may look like. Another possible scenario is

[6] Consider s = "abcbabcbaccba" and $n = 12$. Then on the one hand, `get` s = "abc", but on the other hand, `map` (s !!) (`get` $[0..n]$) = `map` (s !!) $[0..n] = s$.

a more radical paradigm shift. Microsoft's LINQ project, which directly builds on Haskell work, demonstrates that this is not as absurd a prospect as it may seem [19].

And Haskell does not even represent the upper end of the expressiveness spectrum of type systems. Currently, one could say that the object-oriented research community looks towards Haskell when seeking a yardstick of how strong types could be [35,6]. But there is more to discover beyond Haskell. For example, *dependent types* in languages like Coq [2], Epigram [18], and Agda [22] capture semantically richer properties of functions, and could thus lead to new heights in expressiveness for applications in the spirit of free theorems.

References

1. F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.
2. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004.
3. G.E. Blelloch. Prefix sums and their applications. In J.H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 35–60. Morgan Kaufmann, 1993.
4. S. Böhme. Much ado about two. Formal proof development. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/MuchAdoAboutTwo.shtml`, 2007.
5. R.P. Brent and H.T. Kung. The chip complexity of binary arithmetic. In *ACM Symposium on Theory of Computing, Proceedings*, pages 190–200. ACM Press, 1980.
6. M.J. Dominus. Atypical types (Invited talk). At *Object-Oriented Programming, Systems, Languages, and Applications*, 2008.
7. J.P. Fernandes, A. Pardo, and J. Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell Workshop, Proceedings*, pages 95–106. ACM Press, 2007.
8. J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.
9. A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.
10. T. Golding. *Professional .NET 2.0 Generics*. Wrox, 2005.
11. P. Hudak, R.J.M. Hughes, S.L. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *History of Programming Languages, Proceedings*, pages 12-1–12-55. ACM Press, 2007.
12. G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.
13. M.P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, 1995.
14. D.E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.
15. D. Leivant. Polymorphic type inference. In *Principles of Programming Languages, Proceedings*, pages 88–98. ACM Press, 1983.

16. Y.-C. Lin and J.-W. Hsiao. A new approach to constructing optimal parallel prefix circuits with small depth. *Journal of Parallel and Distributed Computing*, 64(1):97–107, 2004.
17. K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *International Conference on Functional Programming, Proceedings*, pages 47–58. ACM Press, 2007.
18. J. McKinna. Why dependent types matter (Invited talk). In *Principles of Programming Languages, Proceedings*, pages 1–14. ACM Press, 2006.
19. E. Meijer. Fundamentalist functional programming (Keynote address). In *Generative Programming and Component Engineering, Proceedings*, page 99. ACM Press, 2008.
20. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML — Revised*. MIT Press, 1997.
21. M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly, 2006.
22. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
23. M. Odersky. The Scala experiment: Can we provide better language support for component systems? (Invited talk). In *Principles of Programming Languages, Proceedings*, pages 166–167. ACM Press, 2006.
24. B. O'Sullivan, D.B. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly, 2008.
25. S.L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
26. J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.
27. M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.
28. J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(6):226–231, 1960.
29. C. Strachey. Fundamental concepts in programming languages. Lecture notes for a course at the *International Summer School in Computer Programming*, 1967. Reprint appeared in *Higher-Order and Symbolic Computation*, 13(1–2):11–49, 2000.
30. J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *International Conference on Functional Programming, Proceedings*, pages 124–132. ACM Press, 2002.
31. J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008.
32. J. Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, 2008.
33. J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.
34. P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.
35. P. Wadler. Faith, evolution, and programming languages: from Haskell to Java to Links (Invited talk). In *Object-Oriented Programming, Systems, Languages, and Applications, Proceedings*, page 508. ACM Press, 2006.
36. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages, Proceedings*, pages 60–76. ACM Press, 1989.