Chapter 1

Generic proofs for combinator-based generic programs

Fermín Reig[1]

Abstract: Generic programming can bring important benefits to software engineering. In particular, it reduces the burden of verification, since generic proofs can be instantiated at many types. Reasoning about programs that use type classes does not enjoy the benefits of generic reasoning, as it requires providing proofs for an arbitrary number of type instances. This makes the process very impractical. We describe a useful method to reason about a class of programs that use type classes, based on the idea that generic functions implemented using overloading can also be expressed polytypically. We demonstrate the method on functions from the 'scrap-your-boilerplate' library, a collection of combinators for generic programming that has been proposed and implemented recently.

1.1 INTRODUCTION

Generic programming enables concise definitions of functions over many types. In the case of parametric polymorphism, the code of the function is the same at every type. In the case of polytypism, the code is different at each type, but it can be derived mechanically from a single polytypic definition (Jeuring 1996; Hinze 2000a).

If similar functions have to be written over and over again for many types, then source code becomes larger. Programming errors increase with program size, and so does the cost of verifying, maintaining and documenting code. Two important benefits of generic programming are smaller programs and smaller proofs.

Two systems for generic programming have been developed recently. In Generic

[1]School of Computer Science and I.T., University of Nottingham.

Haskell (Hinze 2000b), generic functions are defined by induction on the structure of types. To define a generic function, the programmer provides cases for basic types and for structured types built from sums and products. From this information, a compiler or specialiser can generate instances of the generic function for arbitrary types. Here is an example:

$$
\begin{array}{lll}
add\langle Bool\rangle & & = (\vee) \\
add\langle Int\rangle & & = (+) \\
add\langle A\times B\rangle\ (x,y) & (x',y') & = (add\langle A\rangle\ x\ x', add\langle B\rangle\ y\ y') \\
add\langle A+B\rangle\ (inl\ x) & (inl\ y) & = inl\ (add\langle A\rangle\ x\ y) \\
add\langle A+B\rangle\ (inr\ x) & (inr\ y) & = inr\ (add\langle B\rangle\ x\ y) \\
add\langle A+B\rangle\ \_ & \_ & = error\ \texttt{"shape mismatch"}
\end{array}
$$

This function does point-wise addition of two structures (lists, trees, matrices,...) of the same type and shape. The structures may contain booleans and integers. This style of generic programming has been implemented in Generic Haskell (Hinze and Jeuring 2003; Löh 2004) and in Clean (Alimarine and Plasmeijer 2001). Because generic functions take types as parameters in this style, they are also called type-indexed functions. The second proposal, the 'scrap-your-boilerplate' library, (Lämmel and Peyton Jones 2003, 2004) provides a collection of combinators for traversing values of arbitrary datatypes, together with combinators to simulate dynamic type case. A function similar to the one given above can be written like this:

$$add\ x\ y = gzipWith\ ((mkTT\ (+)\ `extTT`\ (mkTT\ (\vee))))\ x\ y$$

Here, *gzipWith* pairs matching values in x and y, and fails if the shapes are different. Then, a dynamic type check on matching values is performed, followed by 'or', addition, or failure if the corresponding types are respectively both boolean, both integer, or something else.

The two approaches support different styles of generic programming and are not comparable in terms of the class of generic functions that can be defined.

Little is known about the formal properties of the 'boilerplate' combinators. Lämmel and Peyton Jones (2003) state the following laws about the combinators *gmapT* and *gmapQ*:

$$
\begin{array}{ll}
gmapT\ id & = id \\
(gmapT\ t)\ \circ (gmapT\ t') & = gmapT\ (t\ \circ t') \\
(gmapQ\ q)\circ (gmapT\ t) & = gmapQ\ (q\circ t)
\end{array}
$$

Since it is not explained in the paper, we can interpret this in two ways. One is that their implementation satisfies these laws. The other one is that any implementation of these functions is required to satisfy them. In either case, they provide no proof that their implementation satisfies them. But there is a good reason for this omission: their library is built using Haskell type classes, and there are currently no established methods for reasoning about overloaded functions in

Haskell. Class functions share only a common type schema, but the implementation (and any laws that it might satisfy) at two different types need have no relation. It would be possible to prove the laws for specific instances, but this does not scale. What we really need are *generic* proofs that cover all possible instantiations.

In this paper, we give generic proofs of the above laws. The insight is that *gmapT* and *gmapQ* can be defined polytypically, instead of using type classes. We implement them in Generic Haskell, which supports proofs about generic functions (by induction on the structure of the type cases) (Hinze 2000a). In addition, we *calculate* a condition for the fusion of traversals that use more complex combinators than *gmapT* and *gmapQ*. This result can be used (by programmers or compilers) to transform code for improved efficiency. To our knowledge, this represents the first published account of formal reasoning about programs that use the 'boilerplate' combinators.

1.2 'BOILERPLATE' COMBINATORS AS GENERIC FUNCTIONS

The 'boilerplate' library is of help when writing programs that do traversals of data built from elaborate, mutually-recursive types. By using a few simple combinators, programmers can avoid writing most of the traversal code by hand. The approach is based on basic combinators that implement one-level, non-recursive traversals. From them, it is possible to implement more complex traversals, such as exhaustive top-down or bottom-up traversal. There are combinators for transforming values and for querying them; and there are purely functional and monadic versions of the combinators.

The basic combinators are *gmapT*, which applies a generic transformation to the top-level children of a value, and *gmapQ*, which applies a generic query to the top-level children of a value and returns a list of the results. For instance, let x be the value $(\texttt{'a'},(\texttt{'b'},\texttt{'c'}))$ and t the transformation that converts characters to upper case and is the identity transformation at any other type. Then *gmapT t x* evaluates to $(\texttt{'A'},(\texttt{'b'},\texttt{'c'}))$. That is, the mapping is not recursive. (x has two top-level children, `'a'` and the pair $(\texttt{'b'},\texttt{'c'})$; t is the identity on pairs.) Let q be the query that gets the ASCII number of a character and returns -1 for any other type. Then *gmapQ q x* evaluates to the list $[97,-1]$.

Because of the regularity of their behaviour, instances of *gmapT* and *gmapQ* can be generated automatically for every type[2]. Alternatively, *gmapT* and *gmapQ* can be defined as generic functions. By doing so, we gain the possibility of using existing reasoning principles for generic programs (Hinze 2000a). Here is a polytypic implementation of *gmapT*. Later, we use this version to prove the equivalence $(gmapT\ t)\circ(gmapT\ t') = gmapT\ (t\circ t')$.

$$
\begin{array}{lll}
gmapT\langle Unit\rangle & \_\, u & = u \\
gmapT\langle Int\rangle & \_\, i & = i
\end{array}
$$

[2] In the current implementation, the combinator *gfoldl* is generated, and *gmapT* and *gmapQ* are defined in terms of it (Lämmel and Peyton Jones 2004).

$$
\begin{array}{lll}
gmapT\langle A+B\rangle & t\ (inl\ l) & = inl\ (gmapT\langle A\rangle\ t\ l) \\
gmapT\langle A+B\rangle & t\ (inr\ r) & = inr\ (gmapT\langle B\rangle\ t\ r) \\
gmapT\langle A_1\times\cdots\times A_n\rangle & t\ (x_1,...,x_n) & = (t\langle A_1\rangle\ x_1,\ \ ...,\ \ t\langle A_n\rangle\ x_n)
\end{array}
$$

Two comments about this definition are in order. First, note that t, the first argument to *gmapT*, is itself a generic function: it is applied at different types (A_1,..., A_n) in the last equation. Not all implementations of polytypism support first-class generic functions. Hinze's recent version, based on type classes (Hinze 2004), allows them, since generic functions are ordinary Haskell functions[3]. On the other hand, implementations based on translation by specialisation do not support them, although it is the topic of current research (Löh 2004). Second, in languages like Generic Haskell, generic functions are defined by providing type cases for binary sums and products. Before generic functions are applied, a translation based on type isomorphisms takes place (Hinze 2000a). In all the generic functions we have seen so far, the behaviour for n-ary products can be derived from the behaviour for nested binary products. Interestingly, this is not the case for *gmapT* and *gmapQ*: their definition at type $T_1\times\cdots\times T_{n-1}\times T_n$ is different from their definition at type $T_1\times(\cdots\times(T_{n-1}\times T_n)\cdots)$. This is why we have written a case for n-ary products (using an ellipsis as informal syntax). However, supporting n-ary products raises technical difficulties with the types and implementation of generic functions. Fortunately, they are not strictly necessary to implement our combinators. Holdermans et al. (2005) give an alternative translation for Generic Haskell that lets us implement *gmapT* and *gmapQ* describing only the case for binary products. The code would be almost identical; only the last line would need to be changed to the following:

$$gmapT\langle A\times B\rangle\ t\ (x,y) = (gmapT\langle A\rangle\ t\ x,\ \ t\langle B\rangle\ y)$$

In this paper, though, we prefer to use n-ary products, since the resulting definitions correspond more naturally to the English description of the behaviour, namely "apply to all immediate children". If we used binary products only, together with the alternative translation, our proofs would require only minor changes, just like the code.

Here is our polytypic implementation of *gmapQ*. Later, we will use this definition to prove the equivalence $(gmapQ\ q)\circ(gmapT\ t) = gmapQ\ (q\circ t)$.

$$
\begin{array}{lll}
gmapQ\langle Unit\rangle & \_\ \_ & = [\,] \\
gmapQ\langle Int\rangle & \_\ \_ & = [\,] \\
gmapQ\langle A+B\rangle & q\ (inl\ l) & = gmapQ\langle A\rangle\ q\ l \\
gmapQ\langle A+B\rangle & q\ (inr\ r) & = gmapQ\langle B\rangle\ q\ r \\
gmapQ\langle A_1\times\cdots\times A_n\rangle & q\ (x_1,...,x_n) & = [q\langle A_1\rangle\ x_1,\ \ ...,\ \ q\langle A_n\rangle\ x_n]
\end{array}
$$

[3] In fact, Hinze uses Haskell98. First-class generic functions require rank-2 polymorphism, which is not Haskell98. However, popular Haskell implementations support this extension.

1.3 GENERIC PROOFS FOR 'BOILERPLATE' COMBINATORS

Generic functions are defined by induction on the structural representation of types. A suitable proof method for such definitions is fixed-point induction (Hinze 2000a). The main advantage of these proofs is that they are generic, in the sense that a single, type-indexed proof covers all its type instantiations. To construct a proof of a predicate P about generic functions, we first write it as a type-indexed predicate $P\langle T\rangle$. Then, we need to show that it holds for basic types, and that it is preserved by type constructors:

$$
\begin{array}{l}
P\ (Unit) \\
P\ (Int) \\
\forall A\ B.P\ (A) \wedge P\ (B) \Rightarrow P\ (A+B) \\
\forall A_1 \ldots A_n.P\ (A_1) \wedge \ldots \wedge P\ (A_n) \Rightarrow P\ (A_1 \times \cdots \times A_n)
\end{array}
$$

Two additional technical requirement for proofs by fixed-point induction are that $P\langle 0\rangle$ must hold (where 0 is the type that contains $\perp$ as its only element) and that P must be built from equalities and inequalities using universal quantification, conjunction and (finite) disjunction. $P\langle 0\rangle$ is typically satisfied by requiring that functions be strict. (For readers familiar with fixed-point induction, these requirements guarantee that P is a pointed and chain-complete relation (Hinze 2000a).)

1.4 A PROOF OF THE FUSION LAW FOR *gmapT*

This law says that two consecutive one-level traversals can be fused into one.

Theorem 1.1. $\langle \forall t\ t' :: (gmapT\ t) \circ (gmapT\ t') = gmapT\ (t \circ t')\rangle$

This law allows a program transformation that reduces the number of traversals over a data structure. We prepare for the proof by stating the fusion law as a type-indexed predicate, i.e. with explicit type parameters in applications of generic functions:[4]

$$\langle \forall t\ t' :: (gmapT\langle T\rangle\ t) \circ (gmapT\langle T\rangle\ t') = gmapT\langle T\rangle\ (t \circ t')\rangle$$

We now have to show that this holds for all possible types T. The base case are primitive types.

• **Case** $T = Unit \mid Int$

$$
\begin{array}{ll}
 & gmapT\langle T\rangle\ t\ (gmapT\langle T\rangle\ t'\ x) \\
= & \{\ \text{definition of } gmapT\langle Unit \mid Int\rangle\ \} \\
 & gmapT\langle T\rangle\ t\ x \\
= & \{\ \text{definition of } gmapT\langle Unit \mid Int\rangle\ \}
\end{array}
$$

[4] In Haskell a type parameter is still there, but as an implicit parameter in the form of a class dictionary (Augustsson 1993).

$$
\begin{array}{ll}
& x \\
= & \{ \text{ definition of } gmapT\langle Unit \,/\, Int\rangle \ \} \\
& gmapT\langle T\rangle\ (t \circ t')\ x
\end{array}
$$

The induction steps are sums and products. For sums, we need to invoke the induction hypothesis. (We only show the case for *inl*; the case for *inr* is symmetric.)

• **Case** $T = A + B, x = inl\ a$

$$
\begin{array}{ll}
& gmapT\langle T\rangle\ t\ (gmapT\langle T\rangle\ t'\ x) \\
= & \{ \text{ definition of } gmapT\langle A+B\rangle \text{ , } x = inl\ a \ \} \\
& gmapT\langle T\rangle\ t\ (inl\ (gmapT\langle A\rangle\ t'\ a)) \\
= & \{ \text{ definition of } gmapT\langle A+B\rangle \ \} \\
& inl\ (gmapT\langle A\rangle\ t\ (gmapT\langle A\rangle\ t'\ a)) \\
= & \{ \text{ induction } \} \\
& inl\ (gmapT\langle A\rangle\ (t \circ t')\ a) \\
= & \{ \text{ definition of } gmapT\langle A+B\rangle \text{ , } x = inl\ a \ \} \\
& gmapT\langle T\rangle\ (t \circ t')\ x
\end{array}
$$

For tuples, we don't even need the induction hypothesis; we only make use of the definitions of *gmapT* and the function composition operator.

• **Case** $T = A_1 \times \cdots \times A_n, x = (x_1, ..., x_n)$

$$
\begin{array}{ll}
& gmapT\langle T\rangle\ t\ (gmapT\langle T\rangle\ t'\ x) \\
= & \{ \text{ definition of } gmapT\langle A_1 \times \cdots \times A_n\rangle \ \} \\
& gmapT\langle T\rangle\ t\ (t'\langle A_1\rangle\ x_1,\ ...,\ t'\langle A_n\rangle\ x_n) \\
= & \{ \text{ definition of } gmapT\langle A_1 \times \cdots \times A_n\rangle \text{ ; function composition } \} \\
& ((t \circ t')\langle A_1\rangle\ x_1,\ ...,\ (t \circ t')\langle A_n\rangle\ x_n) \\
= & \{ \text{ definition of } gmapT\langle A_1 \times \cdots \times A_n\rangle \ \} \\
& gmapT\langle T\rangle\ (t \circ t')\ x
\end{array}
$$

Finally, we need to check that our predicate holds for the empty type 0. This is the case if *gmapT* is strict.

• **Case** $T = 0, x = \bot$

$$
\begin{array}{ll}
& gmapT\langle T\rangle\ t\ (gmapT\langle T\rangle\ t'\ x) \\
= & \{ \ gmapT \text{ strict, hence } gmapT\langle T\rangle\ t'\ \bot = \bot \ \} \\
& gmapT\langle T\rangle\ t\ \bot \\
= & \{ \ gmapT \text{ strict } \} \\
& \bot \\
= & \{ \ gmapT \text{ strict } \}
\end{array}
$$

$$gmapT\langle T\rangle\ (t \circ t')\ x$$

The other fusion theorem says that a one-level traversal followed by a one-level query can be fused into a one-level query.

Theorem 1.2. $\langle \forall q\ t\ ::\ (gmapQ\ q) \circ (gmapT\ t) = gmapQ\ (q \circ t)\rangle$

The proof proceeds in a similar way to the one shown above. We include it as an appendix. We omit the proof of $gmapT\ id = id$, since it is similar to these two.

In the next section, we give a function over lists and a theorem about that function. We then generalise to a generic function over arbitrary types, and a corresponding theorem. Finally, we construct a generic proof of the theorem.

1.5 A THEOREM ABOUT *occurs*

Consider the following polymorphic function that counts the number of occurrences of a value in a list:

$$\begin{aligned} &occursList \in (Eq\ a) \Rightarrow a \to [a] \to Int \\ &occursList\ v = length \circ filter\ (\equiv v) \end{aligned}$$

(This performs two list traversals. A version that makes only one is easy to write, but the one shown here is slightly more concise; consider it a specification, but one that is executable.) This function satisfies the following theorem:

Theorem 1.3. $\langle \forall x\ xs\ f\ ::\ occursList\ x\ xs \leqslant occursList\ (f\ x)\ (map\ f\ xs)\rangle$

Similar functions for other data structures, such as trees, are likely to be useful as part of a complete library. For each of them we would also state a similar theorem. Even better is to write a single generic function, state a single theorem, and write one proof only. Here is such a generic function, implemented with the 'boilerplate' combinators.

$$\begin{aligned} &occurs\ x\ xs = everything\ (+)\ (0\ `mkQ`\ cnt)\ xs \\ &\quad \textbf{where} \\ &\qquad cnt\ y = \textbf{if}\ y \equiv x\ \textbf{then}\ 1\ \textbf{else}\ 0 \end{aligned}$$

This traverses every node of *xs*, produces an integer for each one of them, and adds them all together. We describe the code briefly, but refer the reader to (Lämmel and Peyton Jones 2003) for the details of the combinators *everything* and *mkQ*. *mkQ* turns functions into generic queries that can be applied to arguments of any type. If a node *n* has the same type as *x*, then *cnt* is applied to *n*; otherwise, the default value 0 is returned. *everything* is defined as follows: $everything\ k\ q\ x = foldl\ k\ (q\ x)\ (gmapQ\ (everything\ k\ q)\ x)$

A generic theorem

The theorem that the generic function satisfies is the following:

Theorem 1.4. $\langle \forall x\, xs\, f :: \mathit{fusable}\,(f) \Rightarrow \mathit{occurs}\, x\, xs \leqslant \mathit{occurs}\,(f\, x)\,(\mathit{everywhere}\,(\mathit{mkT}\, f)\, xs)\rangle$

everywhere does a bottom-up traversal, applying the generic transformation $(\mathit{mkT}\, f)$ to every node of *xs*. Its definition is:

$$\mathit{everywhere}\, t = t \circ \mathit{gmapT}\,(\mathit{everywhere}\, t)$$

mkT makes a generic transformation from a function, that is, it extends the function to arguments of any type. Its specification is:

$(\mathit{mkT}\, f)\, x = f\, x$, if the type of x is the same as the domain of f
$(\mathit{mkT}\, f)\, x = x$, otherwise

The predicate *fusable* says that an *everywhere* transformation followed by an *everything* query, can be performed as a single traversal:

$$\begin{aligned}\mathit{fusable}\,(f) = \langle \forall xs\, q :: \mathit{everything}\,(+)\, q\,(\mathit{everywhere}\,(\mathit{mkT}\, f)\, xs) = \\ \mathit{everything}\,(+)\,(q \circ (\mathit{mkT}\, f))\, xs\rangle\end{aligned}$$

This generalises Theorem 1.2 from one-level traversals to full bottom-up traversals. We believe (but have not proved yet) that this predicate is satisfied for transformations that do not alter the shape of the value *xs*.

The reader may wonder why the generic theorem includes this condition, making it weaker than the specific theorem about lists. The 'boilerplate' implementation of *occurs* is more general and can be used to count occurrences of a value *x* of type *a*, not only in a container of *a*s, but in a structure of arbitrary type. Similarly, an *everywhere* transforamtion is more general than *map* over a container type. For instance, the transformation can remove elements from a container, which *map* is guaranteed not to do. These more general functions satisfy weaker properties.

A generic proof

Our first attempt at a generic proof resulted in a long, complex proof by induction on types. (Just like large programs, long proofs written by hand are more likely to contain errors.) Subsequently, we managed to produce a concise proof where only a few lemmas require proofs by induction on types. (In the proof, we use a version of the code where the function *cnt* has been lambda-lifted because having *x* as a free variable in *cnt* complicates things.)

$$\begin{aligned}&\mathit{occurs}\, x\, xs = \mathit{everything}\,(+)\,(q\, x)\, xs \\ &\quad \textbf{where} \\ &\qquad q\, x = (0\, `\mathit{mkQ}`\, (\mathit{cnt}\, x)) \\ &\qquad \mathit{cnt}\, x = \lambda y \rightarrow \textbf{if}\ x \equiv y\ \textbf{then}\ 1\ \textbf{else}\ 0\end{aligned}$$

We make use of a number of lemmas. The first two say that *gmapQ* and *everything* preserve ordering.

Lemma 1.1. $\langle \forall q\ q' \ ::\ q \mathrel{\dot{\leqslant}} q' \Rightarrow \mathit{gmapQ}\ q \mathrel{\dot{\leqslant}} \mathit{gmapQ}\ q' \rangle$

Lemma 1.2. $\langle \forall q\ q' \ ::\ q \mathrel{\dot{\leqslant}} q' \Rightarrow \mathit{everything}\ (+)\ q \mathrel{\dot{\leqslant}} \mathit{everything}\ (+)\ q' \rangle$

We use the symbol $\dot{\leqslant}$ to denote point-wise ordering between functions: $f \mathrel{\dot{\leqslant}} g \ = \ \langle \forall x :: f\ x \leqslant g\ x \rangle$.

For conciseness, we define the following shorthand. $q'\ f\ x = (q\ (f\ x)) \circ (\mathit{mkT}\ f)$, where q is the query from the **where** clause in the definition of *occurs*. It is used in our last lemma.

Lemma 1.3. $\langle \forall x\ xs\ f \ ::\ q\ x\ xs \ \leqslant \ q'\ f\ x\ xs \rangle$

Proofs of the lemmas are included in an appendix. Finally, with the help of all this, we can give an equational proof of Theorem 1.4 that is simple and short.

$$
\begin{array}{ll}
 & \mathit{occurs}\ x\ xs \\
= & \quad \{ \text{ definition of } \mathit{occurs} \ \} \\
 & \mathit{everything}\ (+)\ (q\ x)\ xs \\
= & \quad \{ \text{ definition of } \mathit{everything} \ \} \\
 & \mathit{foldl}\ (+)\ (q\ x\ xs)\ (\mathit{gmapQ}\ (\mathit{everything}\ (+)\ (q\ x))\ xs) \\
\leqslant & \quad \{ \text{ Lemmas 1.1, 1.2, 1.3 ; definition of } \mathit{foldl} \ \} \\
 & \mathit{foldl}\ (+)\ (q\ x\ xs)\ (\mathit{gmapQ}\ (\mathit{everything}\ (+)\ (q'\ f\ x))\ xs) \\
\leqslant & \quad \{ \text{ Lemma 1.3; definition of } \mathit{foldl} \ \} \\
 & \mathit{foldl}\ (+)\ (q'\ f\ x\ xs)\ (\mathit{gmapQ}\ (\mathit{everything}\ (+)\ (q'\ f\ x))\ xs) \\
= & \quad \{ \text{ definition of } \mathit{everything} \ \} \\
 & \mathit{everything}\ (+)\ (q'\ f\ x)\ xs \\
= & \quad \{ \text{ definition of } q' \ \} \\
 & \mathit{everything}\ (+)\ ((q\ (f\ x)) \circ (\mathit{mkT}\ f))\ xs \\
= & \quad \{ \text{ condition } (\mathit{fusable}\ (f)) \ \} \\
 & \mathit{everything}\ (+)\ (q\ (f\ x))\ (\mathit{everywhere}\ (\mathit{mkT}\ f)\ xs) \\
= & \quad \{ \text{ definition of } \mathit{occurs} \ \} \\
 & \mathit{occurs}\ (f\ x)\ (\mathit{everywhere}\ (\mathit{mkT}\ f)\ xs)
\end{array}
$$

1.6 A FUSION LAW FOR *everywhere*

In the previous sections we have used generic reasoning to prove theorems about generic functions. Here we use it to *calculate* the weakest condition needed for a statement to hold. The statement in question is a generalisation of Theorem 1.1 from a one-layer transformation to a full bottom-up transformation.

Statement. $\langle \forall t\ t' \ ::\ (\mathit{everywhere}\ t) \circ (\mathit{everywhere}\ t') = \mathit{everywhere}\ (t \circ t') \rangle$

As with other fusion laws, its practical importance is that it can be used to optimise programs. When this is done reliably and consistently, programmers are encouraged to write simple functions, and let the compiler turn them into efficient ones. (Like our *occursList* function in § 1.5.)

Alas, it is not the case that two arbitrary *everywhere* transformations can be fused. For instance, let t be a transformation that prepends a fixed character, say `'a'`, to a string and is the identity at other types. It can be defined as $t = mkT\ (\lambda s \to \texttt{'a'} : s)$. Then we have that *everywhere* $(t \circ t)$ applied to the empty string returns `"aa"`, while $(everywhere\ t) \circ (everywhere\ t)$ returns `"aaa"`.[5]

What happens here is that the first transformation creates new nodes where the second transformation is applicable —in our case, new substrings. That is, it changes the structure. A (too strong) condition for fusion is that the first traversal does not change the structure. A weaker condition is that at any nodes that are created or deleted, the second transformation is the identity. What we do in this section is turn this informal description into a precise statement about t and t', via formal calculation. In fact, it is possible that the informal condition is not the weakest one —This is the problem with informal statements. However, the one we calculate is guaranteed to be, by construction, the weakest condition.

The calculation is more interesting than the proofs we have shown so far, in the sense that straightforward induction on the structure of types is not enough to complete the proof.

1.6.1 Calculating a condition for fusion

We proceed as if we were writing a proof, even though we know the statement is not a theorem. Then, when a step cannot be justified, we make it a condition of the statement. The statement is already provable for basic types:

• **Case** $T = Unit \mid Int$

$$
\begin{aligned}
& everywhere\langle T\rangle\ t\ (everywhere\langle T\rangle\ t'\ x) \\
= \quad & \{\ \text{definitions of } everywhere \text{ and } gmapT\langle Unit \mid Int\rangle\ \} \\
& everywhere\langle T\rangle\ t\ (t'\ x) \\
= \quad & \{\ \text{definitions of } everywhere \text{ and } gmapT\langle Unit \mid Int\rangle\ \} \\
& (t \circ t')\langle T\rangle\ x \\
= \quad & \{\ \text{definitions of } everywhere \text{ and } gmapT\langle Unit \mid Int\rangle\ \} \\
& everywhere\langle T\rangle\ (t \circ t')\ x
\end{aligned}
$$

For both sums and products we get, by calculation, the condition that t' commute with $gmapT\ (everywhere\ t)$. This results in the following theorem:

Theorem 1.5. $\langle \forall t\ t' :: (everywhere\ t) \circ (everywhere\ t') = everywhere\ (t \circ t')$

[5] The first application of $(everywhere\ t)$ returns `'a'` $: []$. If `'a'` is prepended, bottom-up, to all nodes of type string in `'a'` $: []$, we get `"aaa"`.

$$
\begin{array}{l}
\Leftarrow \\
\quad t' \circ (gmapT\ (everywhere\ t)) = (gmapT\ (everywhere\ t)) \circ t' \rangle
\end{array}
$$

Let us see the rest of the proof. The case for sums shows that straightforward induction is not sufficient to complete the proof.

• **Case** $T = A + B, x = inl\ a$

$$
\begin{array}{ll}
 & everywhere\langle T\rangle\ (t \circ t')\ x \\
= & \quad \{\ \text{definition of } everywhere\ \} \\
 & (t \circ t')\langle T\rangle\ (gmapT\langle T\rangle\ (everywhere\ (t \circ t'))\ x) \\
= & \quad \{\ \text{definition of } gmapT\langle A+B\rangle\ \} \\
 & (t \circ t')\langle T\rangle\ (inl\ (gmapT\langle A\rangle\ (everywhere\ (t \circ t'))\ a))
\end{array}
$$

At this point in the calculation, in order to proceed towards the goal, we would like to use the fusion law (backwards) to split the traversal $(everywhere\ (t \circ t'))$ into two traversals. But we may not invoke the induction hypothesis as a justification, since here it is *gmapT* that takes a type argument, and our induction hypothesis is about *everywhere*, not *gmapT*. The solution is to strengthen the induction hypothesis and construct a proof by mutual induction. Thus, the new induction hypothesis is:

$$
\begin{array}{l}
\langle \forall t\ t'\ ::\ everywhere\ t \circ everywhere\ t' = everywhere\ (t \circ t') \wedge \\
\qquad gmapT\ (everywhere\ t \circ everywhere\ t') = gmapT\ (everywhere\ (t \circ t'))\rangle
\end{array}
$$

The calculation continues as follows.

$$
\begin{array}{ll}
 & (t \circ t')\langle T\rangle\ (inl\ (gmapT\langle A\rangle\ (everywhere\ (t \circ t'))\ a)) \\
= & \quad \{\ \text{(mutual) induction}\ \} \\
 & (t \circ t')\langle T\rangle\ (inl\ (gmapT\langle A\rangle\ (everywhere\ t \circ everywhere\ t')\ a)) \\
= & \quad \{\ \text{definition of } gmapT\langle A+B\rangle\ \} \\
 & (t \circ t')\langle T\rangle\ (gmapT\langle T\rangle\ (everywhere\ t \circ everywhere\ t')\ x) \\
= & \quad \{\ gmapT \text{ fusion}\ \} \\
 & (t \circ t')\langle T\rangle\ ((gmapT\langle T\rangle\ (everywhere\ t)) \circ (gmapT\langle T\rangle\ (everywhere\ t'))\ x))
\end{array}
$$

This is close to the definition of of $everywhere\ t \circ everywhere\ t'$ except that $t'\langle T\rangle$ should be applied before $(gmapT\langle T\rangle\ (everywhere\ t))$. That these two applications can be swapped is the condition that we require for our theorem to hold.

$$
\begin{array}{ll}
 & (t \circ t')\langle T\rangle\ ((gmapT\langle T\rangle\ (everywhere\ t)) \circ (gmapT\langle T\rangle\ (everywhere\ t'))\ x) \\
= & \quad \{\ \bullet \text{ condition}\ \} \\
 & t\langle T\rangle\ ((gmapT\langle T\rangle\ (everywhere\ t)) \circ t'\langle T\rangle\ \circ (gmapT\langle T\rangle\ (everywhere\ t'))\ x) \\
= & \quad \{\ \text{definition of } everywhere, \text{ twice}\ \} \\
 & everywhere\langle T\rangle\ t\ (everywhere\langle T\rangle\ t'\ x)
\end{array}
$$

The case for $x = inr\ b$ is symmetric.

• **Case** $T = A_1 \times \cdots \times A_n, x = (x1, ..., xn)$

$$
\begin{aligned}
& everywhere\langle T\rangle\ t\ (everywhere\langle T\rangle\ t'\ x) \\
= \quad & \{\ \text{definition of } everywhere\ \} \\
& everywhere\langle T\rangle\ t\ (t'\langle T\rangle\ (gmapT\langle T\rangle\ (everywhere\ t')\ x)) \\
= \quad & \{\ \text{definition of } everywhere\ \} \\
& t\langle T\rangle\ (gmapT\langle T\rangle\ (everywhere\ t)\ (t'\langle T\rangle\ (gmapT\langle T\rangle\ (everywhere\ t')\ x)))
\end{aligned}
$$

Once again, we would like to swap the applications $t'\langle T\rangle$ and $(gmapT\langle T\rangle\ (everywhere\ t))$. We may justify this step by making it a condition of the statement.

$$
\begin{aligned}
& t\langle T\rangle\ (gmapT\langle T\rangle\ (everywhere\ t)\ (t'\langle T\rangle\ (gmapT\langle T\rangle\ (everywhere\ t')\ x))) \\
= \quad & \{\ \bullet \text{ condition}\ \} \\
& (t \circ t')\langle T\rangle\ ((gmapT\langle T\rangle\ (everywhere\ t) \circ gmapT\langle T\rangle\ (everywhere\ t'))\ x) \\
= \quad & \{\ gmapT \text{ fusion}\ \} \\
& (t \circ t')\langle T\rangle\ ((gmapT\langle T\rangle\ (everywhere\ t \circ everywhere\ t'))\ x) \\
= \quad & \{\ \text{definition of } gmapT\langle A_1 \times \cdots \times A_n\rangle\ \} \\
& (t \circ t')\langle T\rangle\ ((evW\ t \circ evW\ t')\langle T_1\rangle\ x_1,\ \ ...,\ \ (evW\ t \circ evW\ t')\langle T_n\rangle\ x_n) \\
= \quad & \{\ \text{induction}\ \} \\
& (t \circ t')\langle T\rangle\ (everywhere\langle T_1\rangle\ (t \circ t')\ x_1,\ \ ...,\ \ everywhere\langle T_n\rangle\ (t \circ t')\ x_n) \\
= \quad & \{\ \text{definition of } gmapT\langle A_1 \times \cdots \times A_n\rangle\ \} \\
& (t \circ t')\langle T\rangle\ (gmapT\langle T\rangle\ (everywhere\ (t \circ t')\ x) \\
= \quad & \{\ \text{definition of } everywhere\ \} \\
& everywhere\langle T\rangle\ (t \circ t')\ x
\end{aligned}
$$

It remains that we prove the second conjunct of the induction hypothesis. This part of the proof is in Appendix E.

1.7 CONCLUSIONS AND FURTHER WORK

The essence of this paper is the idea that reasoning about (a restricted class of) overloaded functions can be practical, provided that those functions can be described in a generic programming style that supports adequate proof methods. When such a translation is not possible (because the function does not admit a generic definition), reasoning is not impossible, but instead of a single generic proof, proofs for every type instance must be provided (van Kesteren et al. 2004; Paulson 2004).

We have looked at functions from the 'boilerplate' library, but the principle is general and is applicable to overloaded functions that can be generated mechanically. (There exists a tool to generate class instances for Haskell (DrIFT).)

Our conviction that proofs are important for generic programs has been strengthened as a result of writing this paper. When we first set out to prove theorems 1.4

and 1.5, we believed that stronger versions of those theorems were true. After all, they looked like straightforward generalisations of known theorems about functions over lists. The confusion can arise because in many cases, *everywhere* computes the same result as a generic *map*. However, *everywhere* is a more general function than a generic *map* —for instance, its application can result in a change of the structure of a container— and satisfies fewer properties. The danger of this sort of mistake is that programmers might perform program transformations (perhaps to improve performance) that are not meaning-preserving.

An interesting area for further work is the application of reasoning to program transformation. A practical disadvantage of generic functions implemented with the 'boilerplate' combinators is the performance penalty caused by dynamic type checks. When enough is known about the types of arguments, it is sometimes possible to replace calls to generic functions by calls to specialised functions. For instance, assume that f has type $Int \to Int$. Then, it is easy to prove that *everywhere* $(mkT\ f)\ xs = map\ f\ xs$ if xs is a list of *Int*s. We also have that *everywhere* $(mkT\ f)\ ys = ys$ if ys is a list of characters, or in general, a value whose type does not contain *Int*s. These transformations eliminate the costs of dynamic type checks and could be performed automatically by a compiler. Being able to reason about generic functions is essential in order to justify the soundness of such transformations.

Acknowlegments. I would like to thank Roland Backhouse for very helpful discussions and feedback. Ralf Lämmel pointed out several errors and suggested improvements to a previous version of the paper. This work is fundend by EPSRC grant GR/S27078/01.

REFERENCES

Alimarine, A. and R. Plasmeijer (2001). A generic programming extension for Clean. In *Implementation of functional languages*, Volume 2312 of *LNCS*, pp. 168–185.

Augustsson, L. (1993, June). Implementing haskell overloading. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pp. 65–73.

DrIFT. `http://repetae.net/john/computer/haskell/DrIFT`.

Hinze, R. (2000a). Generic programs and proofs. Habilitationsschrift, Universität Bonn.

Hinze, R. (2000b, January 19–21,). A new approach to generic functional programming. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, Massachusetts, pp. 119–132.

Hinze, R. (2004). Generics for the masses. In *International Conference on Functional Programming (ICFP 2004)*. ACM Press. ACM SIGPLAN Notices.

Hinze, R. and J. Jeuring (2003). *Generic Programming: Advanced Lectures*, Volume 2793 of *Lecture Notes in Computer Science*, Chapter Generic Haskell: Practice and Theory, pp. 1–56. Springer-Verlag.

Holdermans, S., J. Jeuring, and A. Löh (2005). Generic views on data types. In preparation.

Jeuring (1996). *2nd Int. School on AFP*, Chapter Polytypic programming.

Lämmel, R. and S. Peyton Jones (2003). Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pp. 26–37. ACM Press. ACM SIGPLAN Notices.

Lämmel, R. and S. Peyton Jones (2004). Scrap more boilerplate. In *International Conference on Functional Programming (ICFP 2004)*, pp. ?? ACM Press. ACM SIGPLAN Notices.

Löh, A. (2004). *Exploring Generic Haskell*. Ph. D. thesis, University of Utrecht.

Paulson, L. C. (2004). Organizing numerical theories using axiomatic type classes. *Journal of Automated Reasoning*. To appear.

van Kesteren, R., M. van Eekelen, and M. de Mol (2004). An effective proof rule for general type classes. In *Workshop in Trends of Functional Programming*.

A PROOF OF THE FUSION LAW FOR *gmapQ* AND *gmapT*

$$\langle \forall q\ t\ ::\ (gmapQ\ q) \circ (gmapT\ t) = gmapQ\ (q \circ t) \rangle$$

We write the law as a type-indexed property and prove it by fixed-point induction.

$$\langle \forall q\ t\ ::\ (gmapQ\langle T\rangle\ q) \circ (gmapT\langle T\rangle\ t) = gmapQ\langle T\rangle\ (q \circ t) \rangle$$

- **Case** $T = Unit \mid Int$

$$\begin{array}{ll} & gmapQ\langle T\rangle\ q\ (gmapT\langle T\rangle\ t\ x) \\ = & \quad \{ \text{ definition of } gmapT\langle Unit \mid Int\rangle \ \} \\ & gmapQ\langle T\rangle\ q\ x \\ = & \quad \{ \text{ definition of } gmapQ\langle Unit \mid Int\rangle \ \} \\ & [\,] \\ = & \quad \{ \text{ definition of } gmapQ\langle Unit \mid Int\rangle \ \} \\ & gmapQ\langle T\rangle\ (q \circ t)\ x \end{array}$$

- **Case** $T = A + B, x = inl\ a$

$$\begin{array}{ll} & gmapQ\langle T\rangle\ q\ (gmapT\langle T\rangle\ t\ x) \\ = & \quad \{ \text{ definition of } gmapT\langle A+B\rangle \ \} \\ & gmapQ\langle T\rangle\ q\ (inl\ (gmapT\langle A\rangle\ t\ a)) \\ = & \quad \{ \text{ definition of } gmapQ\langle A+B\rangle \ \} \\ & gmapQ\langle A\rangle\ q\ (gmapT\langle A\rangle\ t\ a) \\ = & \quad \{ \text{ induction } \} \\ & gmapQ\langle A\rangle\ (q \circ t)\ a \end{array}$$

$=$ { definition of $gmapQ\langle A+B\rangle$ }
$gmapQ\langle T\rangle\ (q\circ t)\ x$

• **Case** $T = A+B, x = inr\ b$
Similar
• **Case** $T = A_1 \times \cdots \times A_n, x = (x1, ..., xn)$

$gmapQ\langle T\rangle\ q\ (gmapT\langle T\rangle\ t\ x)$
$=$ { definition of $gmapT\langle A_1 \times \cdots \times A_n\rangle$ }
$gmapQ\langle T\rangle\ q\ (t\langle A_1\rangle\ x_1,\ ...,\ t\langle A_n\rangle\ x_n)$
$=$ { definition of $gmapQ\langle A_1 \times \cdots \times A_n\rangle$ }
$[(q\circ t)\langle A_1\rangle\ x_1,\ ...,\ (q\circ t)\langle A_n\rangle\ x_n]$
$=$ { definition of $gmapQ\langle A_1 \times \cdots \times A_n\rangle$ }
$gmapQ\langle T\rangle\ (q\circ t)\ x$

Case $T = 0$ is satisfied by requiring that *gmapQ* be strict.

B PROOF OF LEMMA 1.1

$\langle \forall q\ q'\ ::\ q \mathrel{\dot{\leqslant}} q' \Rightarrow gmapQ\ q \mathrel{\dot{\leqslant}} gmapQ\ q' \rangle$

By fixed-point induction on the structure of the type argument.
• **Case** $T = Unit\ /\ Int$

$gmapQ\langle T\rangle\ q\ x$
$=$ { definition of $gmapQ\langle Unit\ /\ Int\rangle$ }
$[\,]$
$=$ { definition of $gmapQ\langle Unit\ /\ Int\rangle$ }
$gmapQ\langle T\rangle\ q'\ x$

• **Case** $T = A+B, x = inl\ a$

$gmapQ\langle T\rangle\ q\ x$
$=$ { definition of $gmapQ\langle A+B\rangle$ }
$gmapQ\langle A\rangle\ q\ a$
$\leqslant$ { induction }
$gmapQ\langle A\rangle\ q'\ a$
$=$ { definition of $gmapQ\langle A+B\rangle$ }
$gmapQ\langle T\rangle\ q'\ x$

• **case** $T = A_1 \times \cdots \times A_n, x = (x1, ..., xn)$

$gmapQ\langle T\rangle\ q\ x$

$$
\begin{array}{ll}
= & \{ \text{ definition of } gmapQ\langle A_1 \times \cdots \times A_n \rangle \ \} \\
& [q\langle A_1 \rangle\ x1,\ \ldots,\ q\langle A_n \rangle\ xn] \\
\leqslant & \{ \text{ assumption } \} \\
& [q'\langle A_1 \rangle\ x1,\ \ldots,\ q'\langle A_n \rangle\ xn] \\
= & \{ \text{ definition of } gmapQ\langle A_1 \times \cdots \times A_n \rangle \ \} \\
& gmapQ\langle T \rangle\ q'\ x
\end{array}
$$

We also have to prove the case $T = 0$. It is easy to show that this is satisfied if *gmapQ* is strict, in a similar way as in the proof of the fusion law for *gmapT* in § 1.4.

C PROOF OF LEMMA 1.2

$$\langle \forall q\ q'\ ::\ q \mathrel{\dot{\leqslant}} q' \Rightarrow everything\ (+)\ q \mathrel{\dot{\leqslant}} everything\ (+)\ q' \rangle$$

By fixed-point induction on the structure of the type argument. The proof is by mutual induction using the following induction hypothesis:

$$\begin{array}{l}\langle \forall q\ q'\ ::\ q \mathrel{\dot{\leqslant}} q' \Rightarrow (everything\ (+)\ q \mathrel{\dot{\leqslant}} everything\ (+)\ q' \wedge \\ \qquad\qquad gmapQ\ (everything\ (+)\ q) \mathrel{\dot{\leqslant}} gmapQ\ (everything\ (+)\ q'))\rangle\end{array}$$

D PROOF OF LEMMA 1.3

$$\langle \forall x\ xs\ f\ ::\ q\ x\ xs \leqslant q'\ f\ x\ xs \rangle$$

• **Case** *x* and *xs* have different type.

$$
\begin{array}{ll}
& q\ x\ xs \\
= & \{\ x, xs \text{ have different type; definition of } q\ \} \\
& 0 \\
= & \{\ (f\ x), ((mkT\ f)\ xs) \text{ have different type; definition of } mkQ\ \} \\
& (0\ `mkQ`\ (f\ x))\ ((mkT\ f)\ xs) \\
= & \{ \text{ definition of } q'\ \} \\
& q'\ f\ x\ xs
\end{array}
$$

• **Case** *x* and *xs* have the same type.
We distinguish two cases. When $x \neq xs$, then *x* 'occurs' 0 times in *xs*.

$$
\begin{array}{ll}
& q\ x\ xs \\
= & \{\ x \neq xs; \text{ definition of } q\ \} \\
& 0 \\
\leqslant & \{ \text{ definition of } q'\ \}
\end{array}
$$

$q'\ f\ x\ xs$

If $x = xs$, then x 'occurs' once in xs, and both q and $q'\ f$ return 1.

$$
\begin{array}{ll}
 & q\ x\ xs \\
= & \{\ x = xs\text{; definition of } q\ \} \\
 & 1 \\
= & \{\ \text{definition of } q\ \} \\
 & q\ (f\ x)\ (f\ x) \\
= & \{\ (f\ x) \text{ well typed; definition of } mkT\ \} \\
 & q\ (f\ x)\ ((mkT\ f)\ x) \\
= & \{\ x = xs\text{; definition of } q'\ \} \\
 & q'\ f\ x\ xs
\end{array}
$$

E PROOF OF THEOREM 1.5 (CONTINUED)

Below is the proof, by fixed-point induction, of the second conjunct of the induction hypothesis:

$$
\begin{array}{l}
\langle \forall t\ t'\ ::\ everywhere\ t \circ everywhere\ t' = everywhere\ (t \circ t') \wedge \\
\qquad gmapT\ (everywhere\ t \circ everywhere\ t') = gmapT\ (everywhere\ (t \circ t'))\rangle
\end{array}
$$

- **Case** $T = Unit \mid Int$

$$
\begin{array}{ll}
 & gmapT\langle T\rangle\ (everywhere\ (t \circ t'))\ x \\
= & \{\ \text{definition of } gmapT\langle Unit \mid Int\rangle\ \} \\
 & x \\
= & \{\ \text{definition of } gmapT\langle Unit \mid Int\rangle\ \} \\
 & gmapT\langle T\rangle\ (everywhere\ t \circ everywhere\ t')\ x
\end{array}
$$

- **Case** $T = A + B, x = inl\ a$

$$
\begin{array}{ll}
 & gmapT\langle T\rangle\ (everywhere\ (t \circ t'))\ x \\
= & \{\ \text{definition of } gmapT\langle A + B\rangle\ \} \\
 & inl\ (gmapT\langle A\rangle\ (everywhere\ (t \circ t'))\ a) \\
= & \{\ \text{induction}\ \} \\
 & inl\ (gmapT\langle A\rangle\ (everywhere\ t \circ everywhere\ t')\ a) \\
= & \{\ \text{definition of } gmapT\langle A + B\rangle\ \} \\
 & gmapT\langle T\rangle\ (everywhere\ t \circ everywhere\ t')\ x
\end{array}
$$

- **Case** $T = A + B, x = inr\ b$

Similar

• **Case** $T = A_1 \times \cdots \times A_n, x = (x1, ..., xn)$

$$
\begin{array}{ll}
& \mathit{gmapT}\langle T\rangle\ (\mathit{everywhere}\ t \circ \mathit{everywhere}\ t')\ x \\
= & \quad \{\ \text{definition of } \mathit{gmapT}\langle A_1 \times \cdots \times A_n\rangle\ \} \\
& ((\mathit{evW}\ t \circ \mathit{evW}\ t')\langle T_1\rangle\ x_1,\ \ ...,\ \ (\mathit{evW}\ t \circ \mathit{evW}\ t')\langle T_n\rangle\ x_n) \\
= & \quad \{\ \text{(mutual) induction}\ \} \\
& (\mathit{everywhere}\langle T_1\rangle\ (t \circ t')\ x_1,\ \ ...,\ \ \mathit{everywhere}\langle T_n\rangle\ (t \circ t')\ x_n) \\
= & \quad \{\ \text{definition of } \mathit{gmapT}\langle A_1 \times \cdots \times A_n\rangle\ \} \\
& \mathit{gmapT}\langle T\rangle\ (\mathit{everywhere}\ (t \circ t'))\ x)
\end{array}
$$