Deriving Comparators and Show Functions in Isabelle/HOL

Christian Sternagel and René Thiemann

Institute of Computer Science, University of Innsbruck, Innsbruck, Austria
{christian.sternagel,rene.thiemann}@uibk.ac.at

Abstract. We present an Isabelle/HOL development that allows for the automatic generation of certain operations for user-defined datatypes. Since the operations are defined within the logic, they are applicable for code generation. Triggered by the demand to provide readable error messages as well as to access efficient data structures like sorted trees in generated code, we provide show functions that compute the string representation of a given value, comparators that yield linear orders, and hash functions. Moreover, large parts of the employed machinery should be reusable for other operations like read functions, etc.
In contrast to similar mechanisms, like Haskell's "deriving," we do not only generate definitions, but also prove some desired properties, e.g., that a comparator indeed orders values linearly. This is achieved by a collection of tactics that discharge the corresponding proof obligations automatically.

1 Introduction

Before shedding light on *how* things are handled internally, let us have a look at *what* the new mechanism does by means of an example.

As reasonably simple datatypes consider lists and rose trees

datatype α *list* = *Nil* | *Cons* α (α *list*) datatype α *tree* = *Tree* α (α *tree list*)

where both datatypes store content of type α. Typical operations that are required on specific lists or trees include the following: determine which of two values is smaller, e.g., for sorting; turning a value into a string, e.g., for printing; computing a hash code for a value, e.g., for efficient indexing; etc.

With our development, obtaining such functionality for trees—assuming that it is already available for lists—is as easy as issuing

derive *compare tree* derive *show tree* derive *hashable tree*

which may be read as "derive a compare function for trees, then derive a show function for trees, and finally derive a hash function for trees." Afterwards, we can easily handle sets of trees or dictionaries where trees are used as keys in code generation [3]: comparisons or hash codes are required to invoke the efficient algorithms from Isabelle's collections framework [6] and container library [7].

From the deriving mechanism we get the following functions at our disposal where *order* is a type which consists of the three elements *Eq*, *Lt*, and *Gt*:

$compare :: (\alpha{::}compare)\ tree \Rightarrow \alpha\ tree \Rightarrow order$
$show :: (\alpha{::}show)\ tree \Rightarrow string$
$hashcode :: (\alpha{::}hashable)\ tree \Rightarrow hashcode$

Here, the annotation $\alpha{::}c$ denotes that type variable α has to be in type class c, i.e., trees are comparable ("showable", "hashable") if their node contents are.

This is exactly what one would expect from similar mechanisms like Haskell's `deriving` or Scala's *case classes* (which support automatic definitions of equality checks, show functions, and hash functions).

However, we are in the formal setting of the proof assistant Isabelle/HOL [9], and can thus go one step further and in addition to automatic function definitions also provide automatic proofs of some properties that these functions have to satisfy: since HOL is a logic of total functions, totality is obligatory; for comparators we guarantee that they implement a linear order (see Section 3); and for show functions that they adhere to the *show law* (see Section 4).

Overview. After presenting some preliminaries and related work in Section 2, we first present our two main applications: comparators are the topic of Section 3 and show functions are discussed in Section 4. While we also support the generation of hash functions (without proving any properties about them), we do not discuss them in the remainder, since this would give no further insight.

Afterwards we explain the main parts of the process to generate class instances. Since this is mostly generic, we will present each part with only one of the classes as a leading example. In general the process is divided into the following steps:

1. First, in Section 5, we show how to define the desired operations as recursive functions. To this end, we illustrate a two-level construction principle that guarantees totality.
2. In Section 6, we further illustrate how the defining equations of operations are post-processed for code generation with the aim of improved efficiency.
3. After that, in Section 7, we discuss how to generate proof obligations for the desired properties and use induction along the recursive structure of a datatype to discharge them. Once these properties are proved it is straightforward to derive class instances.

After the explanation of the deriving mechanism, we continue in Section 8 and illustrate how the new infrastructure for comparators can be integrated into existing Isabelle/HOL formalizations. Finally, we conclude in Section 9.

Our formalization is part of the development version of the archive of formal proofs (AFP). Instructions on how to access our formalization as well as details on our experiments are provided at:

`http://cl-informatik.uibk.ac.at/software/ceta/experiments/deriving`

2 Preliminaries and Related Work

Let us start with two remarks about notation: [] and # are syntactic sugar for the list constructors *Nil* and *Cons*, and we use both notations freely; function composition is denoted $\circ$.

Our work is built on top of Isabelle/HOL's new datatype package [1,15], which thus had a strong impact on the specifics of our implementation. Therefore, some more details might be helpful. The new datatype package is based on the notion of *bounded natural functors* (BNFs). A BNF is a type constructor equipped with a map function, set functions (one for each type parameter; collecting all elements of that type which are part of a given value), and a cardinality bound (which is however irrelevant for our purposes). Moreover, BNFs are closed under composition as well as least and greatest fixpoints. At the lowest level, BNF-based (co)datatypes correspond to fixpoint equations, where multiple curried constructors are modeled by disjoint sums (+) of products (×). Finite lists, for example, are obtained as least fixpoint of the equation $\beta = \mathit{unit} + \alpha \times \beta$.

While in principle this might provide opportunity for generic programming *a la* Magalhães et al. [8]—which makes a sum-of-products representation available to the user—there is the severe problem that we not only need to define the generic functions, but also have to prove properties about them. For this reason, we do not work on the low-level representation, but instead access BNFs via its high-level interface, e.g., we utilize the primrec command for specifying primitive recursive functions, and heavily depend on the induction theorems that are generated by the datatype package. A further problem in realizing the approach of [8] is the lack of multi-parameter type classes in Isabelle/HOL. In the following, whenever we mention "primitive recursion," what we actually mean is the specific version of primitive recursion provided by primrec.

Given a type constructor κ with n type parameters $\alpha_1, \ldots, \alpha_n$—written $(\alpha_1, \ldots, \alpha_n)\ \kappa$—the corresponding map function is written map_κ and the set functions set^1_κ, …, set^n_κ, where the superscript is dropped in case of a single type parameter.[1] In the following we will use "datatype" synonymously with "BNF" but restrict ourselves to BNFs obtained as least fixpoints. Moreover, we often use types and type constructors synonymously.

In general we consider arbitrary datatypes of the form

$$\textsf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \kappa = C_1\ \tau_{11} \ldots \tau_{1k_1} \mid \ldots \mid C_m\ \tau_{m1} \ldots \tau_{mk_m} \tag{1}$$

where each τ_{ij} may only consist of the type variables $\alpha_1, \ldots, \alpha_n$, previously defined types, and $(\alpha_1, \ldots, \alpha_n)\ \kappa$ (for which the type parameters $\alpha_1, \ldots, \alpha_n$ may not be instantiated differently; a standard restriction in Isabelle/HOL). In general several mutually recursive datatypes may be defined simultaneously. For simplicity's sake we restrict to the case of a single datatype in our presentation.

Related Work. Our work is inspired by the deriving mechanism of Haskell [10, Chapter 10], which was later extended by Hinze and Peyton Jones [4] as well as Magalhães et al. [8]. Similar functionality is also provided by Scala's[2] case classes

[1] A further technicality—allowing for things like *phantom types*—is the separation between "used" and "unused" type parameters. To simplify matters, we consider all variables to be "used" in the remainder, even though our implementation also supports "unused" ones.

[2] `http://scala-lang.org`

and Janestreet's `comparelib`[3] for OCaml.To the best of our knowledge there is no previous work on a deriving mechanism that also provides formally verified guarantees about its generated functions (apart from our previous work [12,14] on top of the, now old, datatype package that was never described in detail).

However, the basics of generic programming in theorem provers—automatically deriving functions for arbitrary datatypes, but without proofs—where already investigated by Slind and Hurd [11] (thanks to the anonymous referees for pointing us to that work). Not surprisingly, our basic constructions bear a strong resemblance to that work, where the relationship will be addressed in more detail in Section 5.

3 Linear Orders and Comparators

Several efficient data structures and algorithms rely on a linear order of the underlying element type. For example, to uniquely represent sets by lists, they have to be sorted; dictionaries often require a linear order on keys, etc. Hence, in order to use these algorithms within generated code we require linear orders for the corresponding element types.

There are at least two alternative approaches when representing linear orders for some type α. The first one is to provide one (or both) of the orders $<$ or $\leq$, and the second one is to demand a comparator of type $\alpha \Rightarrow \alpha \Rightarrow \mathit{order}$. In the following, we favor the approach using comparators for several reasons:

The first one is related to simplicity. When constructing comparators, only one function has to be synthesized, whereas linear orders in Isabelle/HOL require both of $<$ and $\leq$. Of course we could just synthesize one of those, say $<$, and then define the other using the built-in equality, i.e., $x \leq y$ iff $x < y \vee x = y$. But then even a single invocation of $\leq$ might result in two comparisons.

Concerning efficiency, for some algorithms, one has to invoke two comparisons of elements of type α, where a comparator only needs one. For example, when traversing a binary search tree, it may require two comparisons to figure out, whether we have to go on left, or right, or whether we are already at the right node. Similarly, also when generating linear orders for complex types, we might want to define the lexicographic order on pairs, where we again may need to perform two comparisons between the first entries of the pairs. In contrast, for both examples we get all the required information from one invocation of the comparator. This may lead to an exponential difference when comparing two tree-shaped values. Despite these benefits of comparators we do not want to hide that there are also some disadvantages. For example for numeral types, where $<$ and $\leq$ can be seen as built-in functions, there might be some overhead when computing the full comparison result of a comparator (which needs two comparisons), where a single invocation of $<$ might have been sufficient as in a sorting algorithm.

The last and perhaps most important reason for preferring comparators is the fact that using comparators as a new *dedicated* class for sorting, etc., one does not

[3] `https://github.com/janestreet/comparelib`

interfere with the remaining formalization. The problem here is, that Isabelle's type class for orders allows to specify exactly *one* order $<$. As an example, in IsaFoR [13] we defined $<$ on positions (of terms) as the standard prefix order, which is the natural choice for a large part of the whole formalization, except for sorting. Still we can invoke a sorting algorithm via the comparator for positions, since the orders within the classes "comparator" and "order" may differ.

In order to formalize comparators in Isabelle/HOL, we started by defining a predicate $\mathit{is\text{-}cmp} :: \alpha\ \mathit{comparator} \Rightarrow \mathit{bool}$ that demands three crucial properties for symmetry, equality, and transitivity.

$$\begin{aligned} \mathit{invert\text{-}order}\ (c\ x\ y) &= c\ y\ x \\ c\ x\ y = \mathit{Eq} &\Longrightarrow x = y \\ c\ x\ y = \mathit{Lt} &\Longrightarrow c\ y\ z = \mathit{Lt} \Longrightarrow c\ x\ z = \mathit{Lt} \end{aligned} \tag{2}$$

Here, $\alpha\ \mathit{comparator}$ is a type abbreviation for $\alpha \Rightarrow \alpha \Rightarrow \mathit{order}$, and $\mathit{invert\text{-}order} :: \mathit{order} \Rightarrow \mathit{order}$ swaps Lt with Gt.

We further provide definitions to switch between comparators and linear orders, which eases the integration of our results in the existing Isabelle/HOL infrastructure which primarily works on linear orders, when it comes to sorting, etc, cf. Section 8.

$$\begin{aligned} \mathit{comparator\text{-}of}\ x\ y &= (\mathsf{if}\ x < y\ \mathsf{then}\ \mathit{Lt}\ \mathsf{else}\ \mathsf{if}\ x = y\ \mathsf{then}\ \mathit{Eq}\ \mathsf{else}\ \mathit{Gt}) \\ \mathit{le\text{-}of\text{-}comp}\ c\ x\ y &= (\mathsf{case}\ c\ x\ y\ \mathsf{of}\ \mathit{Gt} \Rightarrow \mathit{False} \mid \_ \Rightarrow \mathit{True}) \\ \mathit{lt\text{-}of\text{-}comp}\ c\ x\ y &= (\mathsf{case}\ c\ x\ y\ \mathsf{of}\ \mathit{Lt} \Rightarrow \mathit{True} \mid \_ \Rightarrow \mathit{False}) \end{aligned}$$

It was an easy exercise to prove that $\mathit{is\text{-}cmp}\ c$ implies that $\mathit{le\text{-}of\text{-}comp}\ c$ and $\mathit{lt\text{-}of\text{-}comp}\ c$ satisfy the conditions of Isabelle/HOL's class for linear orders, and vice versa, if $\leq$ and $<$ form a linear order, then also $\mathit{is\text{-}cmp}\ \mathit{comparator\text{-}of}$.

We further defined a type class for comparators, *compare*, which demands a constant $\mathit{compare} :: \alpha\ \mathit{comparator}$ that satisfies $\mathit{is\text{-}cmp}\ \mathit{compare}$.

In order to define comparators for datatypes, we rely on an auxiliary function that combines a list of elements of type *order* lexicographically.

Definition 1. *We define the function* $\mathit{comp\text{-}lex} :: \mathit{order}\ \mathit{list} \Rightarrow \mathit{order}$ *as*

$$\begin{aligned} \mathit{comp\text{-}lex}\ [\,] &= \mathit{Eq} \\ \mathit{comp\text{-}lex}\ (x\ \#\ \mathit{xs}) &= (\mathsf{case}\ x\ \mathsf{of}\ \mathit{Eq} \Rightarrow \mathit{comp\text{-}lex}\ \mathit{xs} \mid \_ \Rightarrow x) \end{aligned}$$

Now comparators for lists and trees are easily defined. We just compare the constructors first, and in case of equality recurse and combine the results for each argument via *comp-lex*.

Example 2. Since both lists and trees have one type variable α, the corresponding comparators require a comparator $c :: \alpha\ \mathit{comparator}$ as additional argument.

Hence we will define the functions cmp_{list} and cmp_{tree} of types α *comparator* ⇒ α *list comparator* and α *comparator* ⇒ α *tree comparator*, respectively.

$$\begin{aligned}
cmp_{list}\ c\ Nil\ Nil &= comp\text{-}lex\ [\,] \\
cmp_{list}\ c\ Nil\ (Cons\ \_\ \_) &= Lt \\
cmp_{list}\ c\ (Cons\ \_\ \_)\ Nil &= Gt \\
cmp_{list}\ c\ (Cons\ x\ xs)\ (Cons\ y\ ys) &= comp\text{-}lex\ [c\ x\ y, cmp_{list}\ c\ xs\ ys] \qquad (3) \\
cmp_{tree}\ c\ (Tree\ x\ xs)\ (Tree\ y\ ys) &= comp\text{-}lex\ [c\ x\ y, cmp_{list}\ (cmp_{tree}\ c)\ xs\ ys]
\end{aligned}$$

Both comparators are constructed following a general schema which will be discussed further in Section 5, and which produces a comparator cmp_κ of type α_1 *comparator* ⇒ ... ⇒ α_n *comparator* ⇒ $(\alpha_1, \ldots, \alpha_n)\ \kappa$ *comparator.*

4 Show

A show function for type α provides a string representation of any given value of that type, i.e., $show :: \alpha \Rightarrow string$. In order to allow for constant time concatenation of results (and thus avoid unnecessary performance regression) the actual transformation into a string is postponed as long as possible. This is achieved by the usual trick of using functions of type *string* ⇒ *string* (which we will abbreviate to *shows*) instead of plain strings. Then *show* from above is generalized to $shows :: \alpha \Rightarrow shows$. The original show function is easily recovered by $show\ x = shows\ x\ [\,]$.

In our implementation this is further extended by a *nat* argument representing the "precedence" of the context in which the show function is used, providing more flexibility with respect to parenthesization. For simplicity's sake we omit this detail in the following.

Another quirk that is required by user convenience is a special show function for lists of αs, $shows\text{-}list :: \alpha\ list \Rightarrow shows$. E.g., we usually want lists of characters to be printed as string, i.e., "*abc*" instead of "[*a*, *b*, *c*]." In addition, show functions are required to satisfy the *show law*:

$$shows\ x\ (ys\ @\ zs) = shows\ x\ ys\ @\ zs$$

Together this brings us to the type class *show*:

```
class show =
  fixes shows :: α ⇒ shows and shows-list :: α list ⇒ shows
  assumes shows x (ys @ zs) = shows x ys @ zs and
    shows-list xs (ys @ zs) = shows-list xs ys @ zs
```

The show law. One way of looking at the show law is that show functions do not temper with or depend on output produced so far. To see this, consider the specific instance $shows\ x\ ([\,]\ @\ zs) = shows\ x\ [\,]\ @\ zs$ and observe that this requires *shows* always to behave as if called with [ ] as second argument.

Our goal is now to automatically derive a show function for a given datatype.

Example 3. Assuming a show function for list elements s, this would look as follows for the list datatype:

$$\begin{aligned}&\mathit{shows}_{\mathit{list}}\ s\ \mathit{Nil} = \lambda\text{“}\mathit{Nil}\text{”}\\&\mathit{shows}_{\mathit{list}}\ s\ (\mathit{Cons}\ x\ \mathit{xs}) = \lambda\text{“}(\mathit{Cons}\text{”} \circ \textvisiblespace \circ s\ x \circ \textvisiblespace \circ \mathit{shows}_{\mathit{list}}\ s\ \mathit{xs} \circ \lambda\text{“})\text{”}\end{aligned}$$

Here we use two notational conveniences: λ*“text”* is the show function producing the literal string *“text”*; and $\textvisiblespace$ is a show function producing a single space.

For the tree datatype from the introduction it would be:

$$\begin{aligned}&\mathit{shows}_{\mathit{tree}}\ s\ (\mathit{Tree}\ x\ \mathit{ts}) =\\&\quad \lambda\text{“}(\mathit{Tree}\text{”} \circ \textvisiblespace \circ s\ x \circ \textvisiblespace \circ \mathit{shows}_{\mathit{list}}\ (\mathit{shows}_{\mathit{tree}}\ s)\ \mathit{ts} \circ \lambda\text{“})\text{”}\end{aligned}$$

The underlying general schema (Section 5) produces a show function shows_κ of type $(\alpha_1 \Rightarrow \mathit{shows}) \Rightarrow \ldots \Rightarrow (\alpha_n \Rightarrow \mathit{shows}) \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \kappa \Rightarrow \mathit{shows}$.

5 Internal Constructions

In general we have to consider an arbitrary datatype (1) for which we want to define some function

$$f_\kappa :: (\alpha_1 \Rightarrow \sigma_1) \Rightarrow \ldots \Rightarrow (\alpha_n \Rightarrow \sigma_n) \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \kappa \Rightarrow \sigma'$$

that is parameterized by corresponding functions for type parameters and relies on the existence of a function $f_{\kappa'}$ for each datatype κ' that was used in the construction of κ. For comparators and show functions this specializes to

$$\begin{aligned}&\mathit{cmp}_\kappa :: (\alpha_1\ \mathit{comparator}) \Rightarrow \ldots \Rightarrow (\alpha_n\ \mathit{comparator}) \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \kappa\ \mathit{comparator}\\&\mathit{shows}_\kappa :: (\alpha_1 \Rightarrow \mathit{shows}) \Rightarrow \ldots \Rightarrow (\alpha_n \Rightarrow \mathit{shows}) \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \kappa \Rightarrow \mathit{shows}\end{aligned}$$

whose definitions will rely on a comparator (show function) for each of the α_i as well as each κ' used in the definition of κ.

Note that for such κ', the function $f_{\kappa'}$ itself takes arguments for the type parameters of κ'. Thus, for any occurrence $(\tau_1, \ldots, \tau_k)\ \kappa'$ there will be a subterm of the shape $f_{\kappa'}\ g_1 \ldots g_k$, where each g_i depends on the structure of τ_i. For nested recursive datatypes this may result in κ occurring inside a type parameter position of κ', e.g., the rose tree type makes a nested occurrence inside α *tree list*.

Before we discuss this any further, let us have a look at the specification mechanisms of Isabelle/HOL that would in principle support the automatic definition of a function f_κ (like $\mathit{cmp}_{\mathit{tree}}$ or $\mathit{shows}_{\mathit{tree}}$)

One candidate would be Isabelle's function package [5] by Krauss. This would require automatic termination proofs, and recursion through previously defined $f_{\kappa'}$ is only possible after the automatic generation of congruence rules, which both seems at least tedious. Krauss himself remarked that the function package might not be the right solution for our purposes (personal communication).

The other candidate is primitive recursion, which is provided by the datatype package in form of the primrec command. When using primrec, termination is obtained automatically in exchange for certain syntactic restrictions, which we will call *primitive recursive form* in the following. Essentially, we may only perform pattern matching on one argument, and for a left-hand side like $g\ (C\ t_1\ \ldots t_n)$, the recursive calls must be of the form $\mathit{maps}\ g\ t_i$ where *maps* is a combination of canonical map functions of those types which are involved in nesting. For instance, if g takes lists as argument, then *maps* is the identity, since the datatype of lists is not nested. If g takes rose trees as argument then *maps* is the map function on lists, since trees are nested within lists.

Note that neither $\mathit{cmp}_{\mathit{list}}$ and $\mathit{cmp}_{\mathit{tree}}$ from Example 2 nor $\mathit{shows}_{\mathit{tree}}$ from Example 3 are in primitive recursive form. It is well-known how to reduce the pattern matching to only one argument, by moving the pattern matching into a case-expression on the right-hand side. In the case of lists we are done: the defining equations are in primitive recursive form, and from these we can easily derive the equations of Example 2 and Example 3. However, in the presence of nesting there is still some work to be done.

For instance, one can apply a nested-to-mutual translation as proposed by Slind and Hurd [11]. We actually applied this definitional principle in our previous version [12,14]. However, it has the disadvantage of not being modular, in the sense that in the presence of nesting we could not reuse existing constants. As an example, in [14] the definition of $\mathit{cmp}_{\mathit{tree}}$ will not contain $\mathit{cmp}_{\mathit{list}}$ itself, but a fresh copy of the definition of $\mathit{cmp}_{\mathit{list}}$, specialized to lists of trees. And even worse, when proving properties of $\mathit{cmp}_{\mathit{tree}}$, the tactic has to reprove all properties for the copy of $\mathit{cmp}_{\mathit{list}}$ and cannot reuse properties of $\mathit{cmp}_{\mathit{list}}$.

In the remainder, we describe another workaround which will establish primitive recursive form w.r.t. primrec, and allow modular proofs. The main problem is that calls like $\mathit{cmp}_{\mathit{list}}\ (\mathit{cmp}_{\mathit{tree}}\ c)\ \mathit{xs}$ and $\mathit{shows}_{\mathit{list}}\ (\mathit{shows}_{\mathit{tree}}\ s)\ \mathit{ts}$ are not of the desired form, as neither $\mathit{cmp}_{\mathit{list}}$ nor $\mathit{shows}_{\mathit{list}}$ is the map function for lists. In general we have to gracefully handle patterns of the shape $f_{\kappa'}\ (f_\kappa\ f)$ (where $f_{\kappa'}$ and f_κ might of course take more than one argument function, but such cases can be handled similarly).

The essential idea is now instead of defining $f_\kappa\ f_1 \ldots f_n\ (C_i\ x_1 \ldots x_n) = \ldots$, to encode the information that is provided by the argument functions f_i already into the type of the first argument $C_i\ x_1 \ldots x_n$ of type $(\alpha_1, \ldots, \alpha_n)\ \kappa$. This is akin to assuming that the f_i have already been partially applied to the appropriate subterms x_i, thus we call such functions *partially applied* (comparator or show) functions and denote them by prefixing the function name with a p. In the following we depict the type changes in the general case as well as for comparators and show functions:

$$\begin{aligned}
\mathit{pf}_\kappa &:: (\sigma_1, \ldots, \sigma_n)\ \kappa \Rightarrow \sigma' \\
\mathit{pcmp}_\kappa &:: (\alpha_1 \Rightarrow \mathit{order}, \ldots, \alpha_n \Rightarrow \mathit{order})\ \kappa\ \ \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \kappa \Rightarrow \mathit{order} \\
\mathit{pshows}_\kappa &:: (\mathit{shows}, \ldots, \mathit{shows})\ \kappa \Rightarrow \mathit{shows}
\end{aligned}$$

Given a partially applied function it is easy to define the originally intended one by using canonical maps:

$$\begin{aligned} f_\kappa\ f_1 \dots f_n &= pf_\kappa \circ map_\kappa\ f_1 \dots f_n \\ cmp_\kappa\ c_1 \dots c_n &= pcmp_\kappa \circ map_\kappa\ c_1 \dots c_n \\ shows_\kappa\ s_1 \dots s_n &= pshows_\kappa \circ map_\kappa\ s_1 \dots s_n \end{aligned}$$

Now let us turn to the construction of such partially applied functions using the primrec mechanism. To ease matters, we provide some auxiliary Isabelle/ML functions (as opposed to HOL functions that can be reasoned about inside the logic). Please keep in mind that in the following we just describe general schemas of putting together certain terms and *not* recursive Isabelle/HOL functions. They are similar to the interpretation $[\![\cdot]\!]_{\Theta,\Gamma}$ of Slind and Hurd [11], but differ since only the former produce terms which fit the requirements of primrec.

Given a type constructor κ together with a function $f_\kappa :: (\alpha_1, \dots, \alpha_n)\ \kappa \Rightarrow \sigma$, we support the construction of what we call a *map block* for type τ

$$\mathcal{M}^f_\tau = \begin{cases} f_\kappa & \text{if } \tau = (\tau_1, \dots, \tau_n)\ \kappa \\ map_{\kappa'}\ \mathcal{M}^f_{\tau_1}\ \dots\ \mathcal{M}^f_{\tau_\ell} & \text{if } \tau = (\tau_1, \dots, \tau_\ell)\ \kappa' \text{ with } \kappa' \neq \kappa \\ \lambda x.\, x & \text{otherwise} \end{cases}$$

The purpose of a map block is to relay recursive calls to f_κ through arbitrary layers of type constructors. Note that this matches exactly the requirements of the primrec command.

Given a function $f_{\kappa'}$ for each $\kappa' \neq \kappa$ occurring in τ, we further support the construction of a corresponding *compose block* for type τ

$$\mathcal{C}^f_\tau = \begin{cases} \lambda x.\, x & \text{if } \tau = (\tau_1, \dots, \tau_n)\ \kappa \\ f_{\kappa'} \circ map_{\kappa'}\ \mathcal{C}^f_{\tau_1}\ \dots\ \mathcal{C}^f_{\tau_\ell} & \text{if } \tau = (\tau_1, \dots, \tau_\ell)\ \kappa' \text{ and } \kappa' \neq \kappa \\ \lambda x.\, x & \text{otherwise} \end{cases}$$

whose purpose is to apply the $f_{\kappa'}$ functions to (via $\mathcal{M}^f_\tau$) appropriately prepared subterms. In this way we can cleanly separate recursive function calls as accepted by primrec from further processing of the corresponding results (via the $f_{\kappa'}$s).

Compose and map blocks are then combined into $\mathcal{C}^f_\tau\ (\mathcal{M}^f_\tau\ x)$ for a variable x of type τ. We illustrate this general construction in the following example.

Example 4. For $\kappa = \mathit{tree}$ and a variable x of type $\tau = \alpha\ \mathit{tree}\ \mathit{list}$ we obtain

$$\mathcal{C}^f_\tau\ \ (\mathcal{M}^f_\tau\ \ x) \ = \ \mathcal{C}^f_\tau\ \ (map\ \ f_\kappa\ \ x) \ = \ (f_{\mathit{list}} \circ map\ \ (\lambda x.\, x))\ \ (map\ \ f_\kappa\ \ x)$$

For comparators with $\tau = (\alpha \Rightarrow \mathit{order})\ \mathit{tree}\ \mathit{list}$ the last term would be

$$pcmp_{\mathit{list}}\ (map\ \ pcmp_{\mathit{tree}}\ x) :: \alpha\ \mathit{tree}\ \mathit{list} \Rightarrow \mathit{order}$$

and for show functions with $\tau = \mathit{shows}\ \mathit{tree}\ \mathit{list}$

$$pshows_{\mathit{list}}\ (map\ \ pshows_{\mathit{tree}}\ x) :: \mathit{shows}$$

which both fit the rules of primrec.

Putting everything together, partial comparators are defined as follows.

$$pcmp_\kappa\ (C_i\ x_1 \dots x_{k_i})\ z = \mathsf{case}\ z\ \mathsf{of}\ C_j\ y_1 \dots y_{k_j}\ \Rightarrow$$

$$\begin{cases} Lt & \text{if } i < j \\ Gt & \text{if } j < i \\ \mathit{comp\text{-}lex}\ [\mathcal{C}^{pcmp}_{\tau_{i1}}\ (\mathcal{M}^{pcmp}_{\tau_{i1}}\ x_1)\ y_1, \dots, \mathcal{C}^{pcmp}_{\tau_{ik_i}}(\mathcal{M}^{pcmp}_{\tau_{ik_i}}\ x_{k_i})\ y_{k_i}] & \text{if } i = j \end{cases}$$

Example 5. For our example types this results in the following definitions:

$$\begin{aligned} &pcmp_{list}\ Nil\ z = (\mathsf{case}\ z\ \mathsf{of}\ Nil \Rightarrow \mathit{comp\text{-}lex}\ []\ |\ Cons\ \_\ \_ \Rightarrow Lt) \\ &pcmp_{list}\ (Cons\ cx\ cxs)\ z = (\mathsf{case}\ z\ \mathsf{of} \\ &\quad Nil \Rightarrow Gt\ |\ Cons\ y\ ys \Rightarrow \mathit{comp\text{-}lex}\ [cx\ y, pcmp_{list}\ cxs\ ys]) \\ &pcmp_{tree}\ (Tree\ cx\ cxs)\ z = (\mathsf{case}\ z\ \mathsf{of} \\ &\quad Tree\ y\ ys \Rightarrow \mathit{comp\text{-}lex}\ [cx\ y, pcmp_{list}\ (map\ pcmp_{tree}\ cxs)\ ys]) \end{aligned}$$

For partial show functions the general schema is

$$\begin{aligned} &shows_\kappa\ (C_i\ x_1 \dots x_{k_i}) = \\ &\quad \lambda\text{“}(C_i\text{”} \circ \text{\textvisiblespace} \circ \mathcal{C}^{pshows}_{\tau_{i1}}\ (\mathcal{M}^{pshows}_{\tau_{i1}}\ x_1) \circ \text{\textvisiblespace} \circ \dots \circ \text{\textvisiblespace} \circ \mathcal{C}^{pshows}_{\tau_{ik_i}}\ (\mathcal{M}^{pshows}_{\tau_{ik_i}}\ x_{k_i}) \circ \lambda\text{“})\text{”} \end{aligned}$$

Example 6. For the type of rose trees this results in the following definition:

$$\begin{aligned} &pshows_{tree}\ (Tree\ s\ ts) = \\ &\quad \lambda\text{“}(Tree\text{”} \circ \text{\textvisiblespace} \circ s \circ \text{\textvisiblespace} \circ pshows_{list}\ (map\ pshows_{tree}\ ts) \circ \lambda\text{“})\text{”} \end{aligned}$$

which is in the desired primitive recursive form.

It eventually remains to prove the equations of Example 2 and Example 3 from these definitions. For this, we mainly demand compositionality of the various map functions, the simplification rules for map functions, the definitions of all participating comparators (or show functions), and the definitions of the partially applied functions. For example, for the comparator of trees we derive the desired equation as follows, where in the step from (7) to (8) we use the compositionality of map_{tree} and map, and we go from (9) to (8) by unfolding both definitions of cmp_{tree} and cmp_{list}.

$$\begin{aligned} &cmp_{tree}\ c\ (Tree\ x\ xs)\ (Tree\ y\ ys) \tag{4} \\ &= pcmp_{tree}\ (map_{\text{tree}}\ c\ (Tree\ x\ xs))\ (Tree\ y\ ys) \tag{5} \\ &= pcmp_{tree}\ (Tree\ (c\ x)\ (map\ (map_{\text{tree}}\ c)\ xs))\ (Tree\ y\ ys) \tag{6} \\ &= \mathit{comp\text{-}lex}\ [c\ x\ y, pcmp_{list}\ (map\ pcmp_{tree}\ (map\ (map_{\text{tree}}\ c)\ xs))\ ys] \tag{7} \\ &= \mathit{comp\text{-}lex}\ [c\ x\ y, pcmp_{list}\ (map\ (pcmp_{tree} \circ (map_{\text{tree}}\ c))\ xs)\ ys] \tag{8} \\ &= \mathit{comp\text{-}lex}\ [c\ x\ y, cmp_{list}\ (cmp_{tree}\ c)\ xs\ ys] \tag{9} \end{aligned}$$

6 Code Equations for Comparators

Recall that our main motivation was to define functions inside the logic which then should become available for code generation. Hence, after having defined comparators as in Example 5, and having proved the equations of Example 2, we just register the latter as code equations. In this way, only comparators will appear in generated code, and the internal construction via the partially applied comparators remains opaque—and for the same reasons, the partially applied show functions will not occur in the generated code.

Still these code equations are not optimal w.r.t. execution time. Especially in languages with eager evaluation, the right-hand side of (3),

$$\mathit{comp\text{-}lex}\ [c\ x\ y, \mathit{cmp}_{\mathit{list}}\ c\ \mathit{xs}\ \mathit{ys}]$$

is problematic. Even if the first comparison $c\ x\ y$ evaluates to Lt or Gt, also the second argument $\mathit{cmp}_{\mathit{list}}\ c\ \mathit{xs}\ \mathit{ys}$ will be evaluated in eager languages.

To avoid this inefficiency, we completely unfold applications of *comp-lex* in the right-hand sides of the equations in Example 2 before handing them over to the code generator. To be more precise, unfolding is always performed w.r.t. the following three equations (which are all easily proved):

$$\begin{aligned}&\mathit{comp\text{-}lex}\ [\,] = \mathit{Eq}\\ &\mathit{comp\text{-}lex}\ [x] = x\\ &\mathit{comp\text{-}lex}\ (x\ \#\ y\ \#\ \mathit{xs}) = (\mathsf{case}\ x\ \mathsf{of}\ \mathit{Eq} \Rightarrow \mathit{comp\text{-}lex}\ (y\ \#\ \mathit{xs})\ |\ z \Rightarrow z)\end{aligned}$$

The advantage of doing this just for code generation is that we can still use properties of *comp-lex* within proofs, e.g., when showing that our comparators really behave like comparators. Moreover, we can keep the canonical structure as described in Example 2 without having to perform lots of case splits.

After the expansion, the right-hand side of the code equation for (3) becomes

$$\mathsf{case}\ c\ x\ y\ \mathsf{of}\ \mathit{Eq} \Rightarrow \mathit{cmp}_{\mathit{list}}\ c\ \mathit{xs}\ \mathit{ys}\ |\ z \Rightarrow z$$

where even in eager languages the recursive call will only be evaluated on demand.

7 Correctness of Generated Functions

Eventually we have to ensure correctness of the generated show functions and comparators. For comparators, this amounts to proving the following soundness theorems for our example types and for the general case, and similar theorems have to be proved regarding the show law.

$$\begin{aligned}&\mathit{is\text{-}cmp}\ c \Longrightarrow \mathit{is\text{-}cmp}\ (\mathit{cmp}_{\mathit{list}}\ c) \qquad \mathit{is\text{-}cmp}\ c \Longrightarrow \mathit{is\text{-}cmp}\ (\mathit{cmp}_{\mathit{tree}}\ c)\\ &\mathit{is\text{-}cmp}\ c_1 \Longrightarrow \ldots \Longrightarrow \mathit{is\text{-}cmp}\ c_n \Longrightarrow \mathit{is\text{-}cmp}\ (\mathit{cmp}_\kappa\ c_1\ \ldots\ c_n)\end{aligned} \tag{10}$$

Although the theorems are clearly sufficient to easily plug together valid comparators, they are not sufficient when proving the soundness theorem for a new datatype which uses nested recursion, such as rose trees. To illustrate the problem, recall the defining equation for $\mathit{cmp}_{\mathit{tree}}$:

$$\mathit{cmp}_{\mathit{tree}}\ c\ (\mathit{Tree}\ x\ \mathit{xs})\ (\mathit{Tree}\ y\ \mathit{ys}) = \mathit{comp\text{-}lex}\ [c\ x\ y, \mathit{cmp}_{\mathit{list}}\ (\mathit{cmp}_{\mathit{tree}}\ c)\ \mathit{xs}\ \mathit{ys}]$$

In order to prove the soundness theorem for cmp_{tree}, we clearly need correctness of cmp_{list}. However, since cmp_{list} is invoked on $cmp_{tree}\ c$, the soundness theorem for cmp_{list} can only be applied if we already would have the soundness theorem for cmp_{tree}, and thus the current form of the soundness theorems is not strong enough in the presence of nesting.

As a solution, we always generate pointwise soundness theorems which are based on pointwise properties of a comparator. From the pointwise theorems we can then easily conclude the soundness theorems stated above.

In detail, for symmetry, transitivity, equality, the show law, etc., we define pointwise variants. Here, we only illustrate transitivity. We define transitivity on the level of *order*, and a pointwise variant on the level of comparators. It imposes a stronger variant of transitivity in comparison to (2), which captures all possible combinations of *Lt* and *Eq*. This is required, as we want to prove transitivity in a standalone way, without having to refer to symmetry or equality.

definition $\mathit{trans\text{-}order} :: \mathit{order} \Rightarrow \mathit{order} \Rightarrow \mathit{order} \Rightarrow \mathit{bool}$ where
$\quad \mathit{trans\text{-}order}\ x\ y\ z \longleftrightarrow$
$\quad\quad (x \neq \mathit{Gt} \longrightarrow y \neq \mathit{Gt} \longrightarrow z \neq \mathit{Gt} \wedge ((x = \mathit{Lt} \vee y = \mathit{Lt}) \longrightarrow z = \mathit{Lt}))$

definition $\mathit{ptrans\text{-}comp} :: \alpha\ \mathit{comparator} \Rightarrow \alpha \Rightarrow \mathit{bool}$ where
$\quad \mathit{ptrans\text{-}comp}\ c\ x \longleftrightarrow (\forall\ y\ z.\ \mathit{trans\text{-}order}\ (c\ x\ y)\ (c\ y\ z)\ (c\ x\ z))$

The former definition is more low-level, but has the advantage of being smoothly combinable with *comp-lex*, independent of any comparator:

lemma *comp-lex-trans*: assumes $\mathit{length}\ xs = \mathit{length}\ ys$ and $\mathit{length}\ ys = \mathit{length}\ zs$
and $\forall\ i < \mathit{length}\ zs.\ \mathit{trans\text{-}order}\ (xs\ !\ i)\ (ys\ !\ i)\ (zs\ !\ i)$
shows $\mathit{trans\text{-}order}\ (\mathit{comp\text{-}lex}\ xs)\ (\mathit{comp\text{-}lex}\ ys)\ (\mathit{comp\text{-}lex}\ zs)$

In combination with the already proved partial transitivity property of cmp_{list}

$$(\bigwedge x.\ x \in \mathit{set}\ xs \Longrightarrow \mathit{ptrans\text{-}comp}\ c\ x) \Longrightarrow \mathit{ptrans\text{-}comp}\ (cmp_{list}\ c)\ xs \tag{11}$$

we can now prove the partial transitivity property for trees in a modular way.

$$(\bigwedge x.\ x \in \mathit{set}_{tree}\ t \Longrightarrow \mathit{ptrans\text{-}comp}\ c\ x) \Longrightarrow \mathit{ptrans\text{-}comp}\ (cmp_{tree}\ c)\ t$$

We first apply induction on t. So let $t = \mathit{Tree}\ x_1\ ts_1$ where we can assume the premise and the induction hypothesis.

$$x \in \mathit{set}_{tree}\ (\mathit{Tree}\ x_1\ ts_1) \Longrightarrow \mathit{ptrans\text{-}comp}\ c\ x \quad \text{for all } x \tag{12}$$

$$t_1 \in \mathit{set}\ ts_1 \Longrightarrow \mathit{ptrans\text{-}comp}\ (cmp_{tree}\ c)\ t_1 \quad \text{for all } t_1 \tag{13}$$

We have to prove $\mathit{ptrans\text{-}comp}\ (cmp_{tree}\ c)\ (\mathit{Tree}\ x_1\ ts_1)$, i.e.,

$$\begin{array}{l} \mathit{trans\text{-}order}\ (cmp_{tree}\ c\ (\mathit{Tree}\ x_1\ ts_1)\ t_2)\ (cmp_{tree}\ c\ t_2\ t_3) \\ (cmp_{tree}\ c\ (\mathit{Tree}\ x_1\ ts_1)\ t_3) \end{array} \tag{14}$$

for all t_2 and t_3. In the general case, at this point we perform a case analysis on both t_2 and t_3, where all of the cases where the three leading constructors are different are easily proved by unfolding the transitivity property followed by simplification. Hence, it remains the interesting case with identical constructors. Let $t_2 = \mathit{Tree}\ x_2\ \mathit{ts}_2$ and $t_3 = \mathit{Tree}\ x_3\ \mathit{ts}_3$. Then, (14) simplifies to

$$\begin{array}{l} \mathit{trans\text{-}order}\ (\mathit{comp\text{-}lex}\ [c\ x_1\ x_2,\ \mathit{cmp}_{\mathit{list}}\ (\mathit{cmp}_{\mathit{tree}}\ c)\ \mathit{ts}_1\ \mathit{ts}_2]) \\ \quad (\mathit{comp\text{-}lex}\ [c\ x_2\ x_3,\ \mathit{cmp}_{\mathit{list}}\ (\mathit{cmp}_{\mathit{tree}}\ c)\ \mathit{ts}_2\ \mathit{ts}_3]) \\ \quad (\mathit{comp\text{-}lex}\ [c\ x_1\ x_3,\ \mathit{cmp}_{\mathit{list}}\ (\mathit{cmp}_{\mathit{tree}}\ c)\ \mathit{ts}_1\ \mathit{ts}_3]) \end{array}$$

and via theorem *comp-lex-trans*, it remains to consider all the comparisons of the arguments of the constructor *Tree* which leads to the following proof obligations.

$$\mathit{ptrans\text{-}comp}\ c\ x_1 \tag{15}$$

$$\mathit{ptrans\text{-}comp}\ (\mathit{cmp}_{\mathit{list}}\ (\mathit{cmp}_{\mathit{tree}}\ c))\ \mathit{ts}_1 \tag{16}$$

Here, (15) is immediately solved by (12) and the simplification rules for *set*. And for (16) we first apply (11), then conclude via the induction hypothesis (13).

The proof for the individual arguments is easily generalized to the generic case and follows a simple schema: whenever we hit some foreign type $(\tau_1, \ldots, \tau_m)\ \kappa$, we use the partial transitivity theorem of cmp_κ and proceed recursively on each τ_i; whenever we hit the comparator under consideration, we apply the induction hypothesis, and whenever we hit a comparator for some type variable, we apply the corresponding premise.

In a similar way, also partial symmetry and equality properties are defined and proved. We separated the three properties and did not define a partial comparator property, as the corresponding proofs are all a little bit different and could not easily be merged into a single one. For example, for transitivity we perform one induction and then do a case analyses on two other elements, whereas for symmetry and equality, a single case analysis suffices.

Having proved the partial properties of a comparator and show function, it is easy to derive the main (global) properties of comparators and show functions, namely soundness theorems in (10) and the show law.

8 Integration into Isabelle/HOL Infrastructure

At this point, we have a machinery to automatically derive various class instances for datatypes. Whereas for hash codes and show functions these mechanisms are immediately applicable, this is not the case for comparators. The reason for the latter is the fact, that comparators are not well supported in the Isabelle distribution, where most algorithms for sorting, search-trees, etc. are defined via class *linorder*, and a combination of $\leq$, $<$, and $=$ is applied. To bridge this gap, we offer three different alternatives.

The first alternative is to bridge everything via *lt-of-comp*, *le-of-comp*, and *comparator-of*. This is done when invoking derive *linorder tree*. This command creates a new class instance for trees, *tree* :: (*linorder*) *linorder*, where the syntax says that if the type parameter α is an instance of *linorder*, then so is α *tree*. Here, $<$ and $\leq$ will be defined as *lt-of-comp* ($\mathit{cmp}_{\mathit{tree}}$ *comparator-of*) and *le-of-comp* (cmp_{tree} *comparator-of*), respectively. With this approach one can easily use all the existing algorithms. For example, we used this approach to generate linear orders for the datatypes of the CAVA LTL model checker [2], without changing a single line in the remaining formalization. However the switch between comparators and orders clearly has a negative impact on efficiency.

The second alternative is to modify the existing algorithms so that they are defined via comparators. Here, we provide an easy solution which performs this change just before code generation. It works as follows. First, we defined a class *compare-order*, which demands that there is a linear order and a comparator *compare*, where the induced orders coincide, i.e., $<$ = *lt-of-comp compare* and $\leq$ = *le-of-comp compare* must hold. Afterwards we provide a method which strengthens the class constraints from *linorder* to *compare-order*, where every two consecutive comparisons are replaced by one comparator invocation with the help of several lemmas of the shape:

(if $x \leq y$ then if $x = y$ then P else Q else R) =
(case *compare* x y of *Eq* $\Rightarrow P$ | *Lt* $\Rightarrow Q$ | *Gt* $\Rightarrow R$)

For example, the standard code equations to lookup the value of some key in a red-black tree, *rbt-lookup* :: (α, β) *rbt* $\Rightarrow \alpha \Rightarrow \beta$ *option*, are

rbt-lookup Empty k = *None*
rbt-lookup (*Branch* c l x y r) k =
(if $k < x$ then *rbt-lookup* l k else if $x < k$ then *rbt-lookup* r k else *Some* y)

but after invoking compare-code (α) *rbt-lookup* they are transformed into:

rbt-lookup Empty k = *None*
rbt-lookup (*Branch* c l x y r) k =
(case *compare* k x of *Eq* $\Rightarrow$ *Some* y | *Lt* $\Rightarrow$ *rbt-lookup* l k | *Gt* $\Rightarrow$ *rbt-lookup* r k)

Note that in the original code equations, α only has to be an instance of *linorder*, whereas the modified version enforces α to be an instance of *compare-order*.

In summary, the second approach is also easily integrated. For example, it suffices to invoke compare-code on all constants *rbt-ins, rbt-lookup, rbt-del, rbt-map-entry, sunion-with, and sinter-with* in order to completely adapt the whole red-black tree implementation to work on comparators. And it suffices to call derive *compare-order list* to make lists an instance of *compare-order*, and similarly for other datatypes. This command internally just combines the soundness lemmas (10) of the comparators to assemble a comparator, and then defines $<$ and $\leq$ via *lt-of-comp* and *le-of-comp*. In this way, we could remove over 600 lines of proofs for manually created linear orders in IsaFoR. Moreover, the change from linear orders to comparators led in theory to a linear speed-up when performing comparisons. To measure the impact in practice, we certified over 4122 termination and complexity proofs that have been produced by various tools during the international termination competition, which the generated code had to validate. Whereas the old code required around 17 minutes for the certification of all proofs, the new version required less than 4 minutes.

However, there remains one disadvantage, namely that an existing class instance might interfere with the instance that derive *compare-order* wants to create. For instance, if the ordering on products is defined to be pointwise, then there is no chance to make *prod* an instance of *compare-order*.

Therefore, the third alternative does not require instances of *compare-order*. Instead, one has to copy those functions which are relevant for code generation, manually integrate comparators, and then perform an equivalence proof. Afterwards, one can reuse all of the theorems without much overhead.

As an example, we again consider red-black trees. Here, we manually adapted the lookup function, and the corresponding equivalence proof is straightforward.

```
primrec rbt-comp-lookup :: α comparator ⇒ (α, β) rbt ⇒ α ⇒ β option where
  rbt-comp-lookup c Empty k = None
| rbt-comp-lookup c (Branch - l x y r) k = (case c k x of
      Lt ⇒ rbt-comp-lookup c l k
    | Gt ⇒ rbt-comp-lookup c r k
    | Eq ⇒ Some y)
```

```
lemma rbt-comp-lookup:
  is-cmp c ⟹ rbt-comp-lookup c = ord.rbt-lookup (lt-of-comp c)
```

Afterwards, a theorem like *map-of-entries*—which required a proof of 66 lines—is easily adapted for comparators via *rbt-comp-lookup* and proved in a single line. Notice that *ord.rbt-sorted* and *ord.rbt-sorted* require the order as a parameter, whereas *rbt-lookup* and *rbt-sorted* implicitly take the order from the type class.

```
lemma map-of-entries: rbt-sorted t ⟹ map-of (entries t) = rbt-lookup t
```

```
lemma comp-map-of-entries: is-cmp c ⟹ ord.rbt-sorted (lt-of-comp c) t
  ⟹ map-of (entries t) = rbt-comp-lookup c t
  using linorder.map-of-entries[OF comparator.linorder] rbt-comp-lookup by metis
```

In this way, as a case study we adapted the whole container framework of Lochbihler to use comparators instead of linear orders. Most of the adaptation was straightforward and just required the insertion of suitable equivalence statements like *rbt-comp-lookup*, and the change from *ord.rbt-lookup* to *rbt-comp-lookup*. Moreover, we could remove over 450 lines within the container framework, where manual constructions for orders (now: comparators) and equality-checking have been replaced by one-line invocations of our generators.

9 Conclusion

We presented a mechanism that allows for the automatic derivation of the following operations for arbitrary user-defined datatypes: comparators, show functions, and hash functions. Our work relies on the canonical map functions and corresponding facts that are provided by Isabelle's new datatype package. We further showed how our work can be integrated into existing formalizations, thereby saving lines of code as well as improving the efficiency of generated code.

Acknowledgments. We thank S. Berghofer, J. Blanchette, L. Bulwahn, F. Haftmann, B. Huffman, A. Krauss, P. Lammich, A. Lochbihler, C. Urban, T. Nipkow, D. Traytel, and M. Wenzel for their valuable support w.r.t. motivating our development, information on the old and new datatype packages, and for answering several Isabelle/ML related questions. We thank the anonymous reviewers for their helpful comments. This work was supported by Austrian Science Fund (FWF) projects P27502 and Y757. The authors are listed in alphabetical order regardless of individual contribution or seniority.

References

1. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: Proc. 5th ITP. LNCS, vol. 8558, pp. 93–110 (2014), doi:10.1007/978-3-319-08970-6_7
2. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.: A fully verified executable LTL model checker. In: Proc. 25th CAV. LNCS, vol. 8044, pp. 463–478 (2013), doi:10.1007/978-3-642-39799-8_31
3. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Proc. 10th FLOPS. LNCS, vol. 6009, pp. 103–117 (2010), doi:10.1007/978-3-642-12251-4_9
4. Hinze, R., Peyton Jones, S.: Derivable type classes. ENTCS 41(1), 5–35 (2001), doi:10.1016/S1571-0661(05)80542-0
5. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. JAR 44(4), 303–336 (2010), doi:10.1007/s10817-009-9157-2
6. Lammich, P., Lochbihler, A.: The Isabelle collections framework. In: Proc. 1st ITP. LNCS, vol. 6172, pp. 339–354 (2010), doi:10.1007/978-3-642-14052-5_24
7. Lochbihler, A.: Light-weight containers for Isabelle: Efficient, extensible, nestable. In: Proc. 4th ITP. LNCS, vol. 7998, pp. 116–132 (2013), doi:10.1007/978-3-642-39634-2_11
8. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for Haskell. SIGPLAN Not. 45(11), 37–48 (2010), doi:10.1145/2088456.1863529
9. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
10. Peyton Jones, S.: The Haskell 98 language. JFP 13(1), 139–144 (2003), doi:10.1017/S0956796803001217
11. Slind, K., Hurd, J.: Applications of polytypism in theorem proving. In: Proc. 16th TPHOLs. LNCS, vol. 2758, pp. 103–119 (2003), doi:10.1007/10930755_7
12. Sternagel, C., Thiemann, R.: Haskell's show-class in Isabelle/HOL. Archive of Formal Proofs (Jul 2014), http://afp.sf.net/entries/Show.shtml
13. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Proc. 22nd TPHOLs. LNCS, vol. 5674, pp. 452–468 (2009), doi:10.1007/978-3-642-03359-9_31
14. Thiemann, R.: Generating linear orders for datatypes. Archive of Formal Proofs (Aug 2012), http://afp.sf.net/entries/Datatype_Order_Generator.shtml
15. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: Proc. 27th LICS. pp. 596–605 (2012), doi:10.1109/LICS.2012.75