Data Refinement in Isabelle/HOL

Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow

Technische Universität München

Abstract. The paper shows how the code generator of Isabelle/HOL supports data refinement, i.e., providing efficient code for operations on abstract types, e.g., sets or numbers. This allows all tools that employ code generation, e.g., Quickcheck or proof by evaluation, to compute with these abstract types. At the core is an extension of the code generator to deal with data type invariants. In order to automate the process of setting up specific data refinements, two packages for transferring definitions and theorems between types are exploited.

1 Introduction

Algorithm verification is most convenient at a high level of abstraction, reasoning about data in terms of sets, functions and other mathematical concepts. However, when running the verified code we want to replace sets and functions by lists and trees, to make them efficiently executable. This replacement is called *data refinement*, and the ideal theorem prover should do this fully automatically once we prove that the concrete representation is adequate.

This paper describes a data refinement framework for Isabelle/HOL that automatically replaces abstract data structures by concrete ones during code generation. Our main contribution is a lightweight infrastructure and methodology that reduces data refinement entirely to code generation, requires zero effort from the user and is based on a minimal extension of the code generator.

More formally, data refinement replaces an *abstract* data type A by a more *concrete* one C in the generated code. The typical example is the implementation of sets by lists. The concrete type is also called the *implementation* or *representation.* Refining A by C requires an *abstraction function* $\mathit{Abs} :: C \to A$ (e.g., mapping $[1, 2]$ to $\{1, 2\}$) and an *invariant* $\mathit{inv} :: C \to \mathit{bool}$ (e.g., ruling out lists with duplicates). The basic picture is shown in Figure 1.

The standard approach is to demand that Abs is a homomorphism: for every operation $f \in \Sigma$ (the primitive operations that need implementing) on the abstract type and its concrete implementation f' it must be shown that $f(\mathit{Abs}(x)) = \mathit{Abs}(f'(x))$. A system supporting data refinement on this basis will require the user to prove the homomorphism property for all operations to ensure soundness of the refinement step. This means that a new trusted component is added to the system, the refinement manager. A typical example for this approach is the KIV system [17].

We turn the approach on its head: Rather than check that the correct homomorphism theorems have been proved before code is generated, the homomorphism theorems themselves are the glue code between f and f'. More precisely,

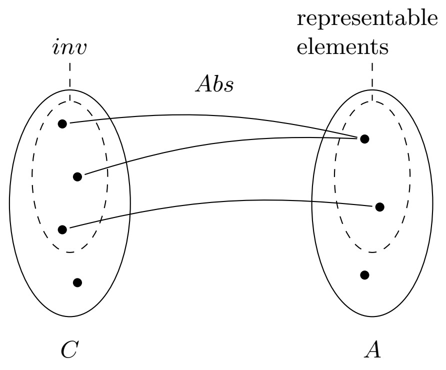

Fig. 1. Data refinement

we instruct the code generator to view A as an algebraic data type with the single (uninterpreted!) constructor $Abs :: C \to A$. Now $f(Abs(x)) = Abs(f'(x))$ is a code equation that performs pattern matching on Abs to turn a call of f into a call of f'. This is the key point of our approach: We generate code for the actual function f, not for some other function f' for which some additional theorems show that it implements f in the correct manner. This form of data refinement is completely automatic: Once a particular refinement of A by C has been set up, generating code involving functions on A involves no further input by the user. This works amazingly well and is explained in §2.

Unfortunately it breaks down once we have a non-trivial invariant and can only prove $inv(c) \Longrightarrow f(Abs(c)) = Abs(f'(c))$. This is a conditional equation and thus unsuitable for generating code. At this point we need to introduce a minimal extension of the code generator that deals with invariants. This is the contents of §3, where the correctness of the extension is also proved.

As a final generalization we allow A to be a nested type expression. This complicates matters and is the subject of §4.

Data refinement is crucial for Isabelle/HOL because it enables code generation for some of the most important types, namely sets and numbers. The details follow, but we can already mention that this is essential for two important applications, in addition to explicit algorithm development by the user: Quickcheck, Isabelle's automatic counterexample search facility [2], and proof by evaluation. Both take advantage of default implementations of sets and numbers to execute seemingly abstract statements like $\{1, 1/2\} \cap \{1 - 1/2, 2\} = \{1/2\}$.

1.1 Code Generation

Isabelle/HOL supports code generation for a number of functional programming languages (SML, OCaml, Haskell, Scala). Basically, equational theorems in HOL, called *code equations*, are translated into function definitions in the target languages. A mathematical treatment of this translation process, including correctness proofs, can be found elsewhere [5]. We stay on the level of code equations here and do not need to worry about the further translation steps. The key correctness property of the generated code is that any evaluation in the target language corresponds to an equality provable in HOL. In other words, the generated code is merely some form of fast rewrite engine which can be used to derive some equations.

When we want to generate code for some function f, any list of equations of the form $f\ \dots = \dots$ (with pattern matching on the lhs) can (in principle) serve as code equations, not just the original definition of f. Thus we are free to define a second, more efficient function g, prove $f(x) = g(x)$, and use this equation (together with the ones for g) as the code equations for f.

Algebraic data types in HOL are turned into equivalent algebraic data types in the target language. Interestingly, the correctness proof revealed that in fact any function in HOL can in principle become a constructor function in the target language (but of course not a defined function at the same time).

1.2 Isabelle/HOL

Isabelle/HOL [16] is based on Church's simple type theory. Types τ are built from type variables (denoted by α, β, ...) and type constructors κ with a fixed arity. The function type is $\rightarrow$ as usual. The notation $t :: \tau$ means that term t has type τ. In concrete examples we use Isabelle/HOL's syntax: $\Rightarrow$ instead of $\rightarrow$ and 'a instead of α. The qualified name $A.f$ refers to function f from theory A. Besides $\forall$, we also use this symbol $\bigwedge$ for universal quantification.

In our examples we employ the usual standard types: lists ('a list), sets ('a set), booleans (bool), and the type of optional values ('a option) with constructors Some and None. The primitive way of introducing new types in Isabelle/HOL is the **typedef** command. It takes a non-empty set comprehension $S = \{x :: \tau.\ P\ x\}$, defines a new type σ, and axiomatizes two isomorphisms $Abs :: \tau \rightarrow \sigma$ and $rep :: \sigma \rightarrow \tau$ as follows: the image of rep is in S (i.e., $\forall x.\ rep\ x \in S$), Abs is a left-inverse for rep (i.e., $\forall x.\ Abs\ (rep\ x) = x$), and rep is a left-inverse for Abs on S (i.e., $\forall x \in S.\ rep\ (Abs\ x) = x$).

2 Basic Data Refinement

We start by considering the situation where there is no invariant. The standard example is the implementation of sets by lists with no restrictions on the order or multiplicity of the elements in the lists. More efficient representations are considered later on, but this one illustrates the basic method well.

The relation between lists and sets is an instance of Figure 1 where $C =$ 'a list, $A =$ 'a set, inv is true everywhere (every list is a valid representation) and $Abs =$ set, a predefined function that returns the set of elements in a list. Infinite sets are not representable (but see the end of the section).

As explained in the Introduction, for code generation purposes we will now consider the abstraction function set as the single constructor of type 'a set. Arbitrary constants (of an appropriate type) can be turned into data type constructors in the generated code (see §1.1). We call such constants *pseudo-constructors*.

Like ordinary constructors, they have no defining code equations but other code equations can use them in patterns on the left-hand side. There are no particular logical properties that such pseudo-constructors have to satisfy—they do not have to be injective or exhaust the abstract type. This is how we instruct the code generator to view set as a constructor:

code_datatype set

Thus in the generated code the type 'a set will become a data type whose elements are in fact lists, but wrapped up in the constructor set. For the primitive set operations we can easily prove alternative equations that pattern-match on set. Here are some examples:

lemma [code]: $\{\} =$ set []
lemma [code]: Set.insert x (set xs) $=$ set (List.insert x xs)
lemma [code]: Set.remove x (set xs) $=$ set (List.removeAll x xs)

The [code] tag tells the code generator that a theorem should be considered a code equation and used instead of the original definition of the function involved.

The technique allows the replacement of one type by another type with surprising ease, based purely on the equational semantics of the code generator.

We now generalize from the example. We assume that $A = (\overline{\alpha})\kappa$, where κ is a type constructor and $\overline{\alpha}$ a list of type variables, its arguments; C is unrestricted. We start by defining $Abs :: C \to A$ and registering it as a pseudo-constructor (via **code_datatype**) in order to pattern-match on it in the code equations for the $f \in \Sigma$. There are no restrictions *per se* on the type of f, but in order to abstract the standard pattern seen in the set/list example we need to make some assumptions on the argument types.

Definition 1. *We call a type $\tau_1 \to \cdots \to \tau_n \to \tau$* basic *(where τ is not a function type) iff all τ_i are either of the form $(\ldots)\kappa$ or do not contain κ.*

We assume that all functions in Σ have a basic type, a property that is satisfied by all our applications. Derived functions can of course have arbitrary types.

Now we must prove for each $f \in \Sigma$ a code equation

$$f\ a_1\ \ldots\ a_n\ =\ t$$

where $a_i = Abs(x_i)$ (if $\tau_i = (\ldots)\kappa$) or $a_i = x_i$ (otherwise). The free variables of t must be contained in $\{x_1, \ldots, x_n\}$.

Now terms involving type κ can be handled by the code generator: the code for all primitive functions f has just been proved, and code for derived functions is generated as always. Hence it must be stressed that the only work we need to do is to prove the code equations for the $f \in \Sigma$.

As described above, all occurrences of type κ are refined by the same type. However, our infrastructure does not by itself enforce this: Lochbihler [12] generalizes our approach to multiple representations. He exploits the fact that there can be multiple pseudo-constructors for any type. In fact, Isabelle's default refinement of sets supports cofinite sets, too, by means of a second pseudo-constructor coset :: 'a list $\Rightarrow$ set where coset $xs = -$ set xs ("$-$" is complement).

3 Data Refinement with Invariants

3.1 Motivation and Example

Implementing sets by lists with possibly repeated elements, as in the previous section, is inefficient. Therefore we now impose the invariant that all elements of the representing lists are distinct and call such lists *distinct lists*. The situation is again the one in Figure 1 with $C =$ 'a list, $A =$ 'a set, $Abs =$ set, but now $inv =$ distinct, a predefined function that tests if all elements of a list are distinct.

But now there is the problem that our pseudo-constructor set can also be applied to lists that are not distinct. As a consequence, some equations for the primitive set operations only hold conditionally, for example

distinct $xs \Longrightarrow$ Set.remove x (set xs) $=$ set (List.remove1 x xs)

This conditional theorem will be rejected as a code equation by the code generator. For soundness reasons the precondition cannot simply be dropped, but without it the theorem does not hold because List.remove1 removes at most one occurrence of x from xs and not all of them like List.removeAll. Our solution is to introduce an intermediate type 'a dlist for distinct lists (see Figure 2). Thus

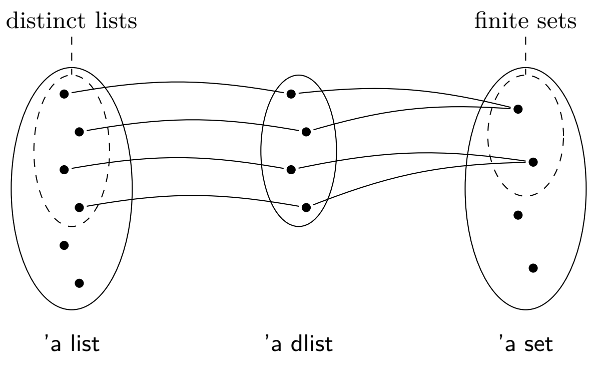

Fig. 2. Sets by distinct lists using 'a dlist

we split the implementation into two steps: the new *subtype* step from 'a list to 'a dlist, where 'a dlist is a new type that is isomorphic to a subset of 'a list, the distinct lists, followed by the basic data refinement of 'a set by 'a dlist which does not involve an invariant anymore and can be dealt with by the method of the previous section.

The new subtype with an invariant is defined by **typedef** (see §1.2):

typedef 'a dlist $= \{xs$::'a list. distinct $xs\}$
  morphisms list Dlist

The **morphisms** directive just renames the canonical *rep* and *Abs* functions to

list :: 'a dlist ⇒ 'a list
Dlist :: 'a list ⇒ 'a dlist

The axiomatization of *rep* and *Abs* in §1.2 can also be presented like this:

$$\mathsf{Dlist}\ (\mathsf{list}\ dxs) = dxs \tag{1}$$

$$\mathsf{distinct}\ xs \longleftrightarrow \mathsf{list}\ (\mathsf{Dlist}\ xs) = xs, \tag{2}$$

Using the two isomorphisms we can define all primitive operations on dlist by lifting corresponding operations on list. For example, this is the definition of Dlist.remove :: 'a ⇒ 'a dlist ⇒ 'a dlist:

$$\mathsf{Dlist.remove}\ x\ dxs = \mathsf{Dlist}\ (\mathsf{List.remove1}\ x\ (\mathsf{list}\ dxs)) \tag{3}$$

Then we bridge the gap between 'a set and 'a dlist by a new pseudo-constructor dset :: 'a dlist ⇒ 'a set:

dset dxs = set (list dxs)

If we assume that we already have all primitive operations on the type 'a dlist together with the necessary properties, it is again straightforward to prove code equations implementing set operations, for example for Set.remove:

lemma [code]: Set.remove x (dset xs) = dset (Dlist.remove x xs)

Therefore we turn to the problem of how to implement dlist operations by list operations. Using Dlist as a pseudo-constructor as in the previous section runs into the same problem as before:

$$\mathsf{distinct}\ xs \Longrightarrow \mathsf{Dlist.remove}\ x\ (\mathsf{Dlist}\ xs) = \mathsf{Dlist}\ (\mathsf{List.remove1}\ x\ xs) \tag{4}$$

is only provable under the assumption distinct xs. Therefore we try the definition of Dlist.remove (3) itself as a code equation. Now we need to execute list on the rhs and face the same problem:

$$\mathsf{list}\ (\mathsf{Dlist}\ xs) = xs \tag{5}$$

is only provable if distinct xs. Therefore we extend the code generator for this special case as follows. Attaching attribute [code abstype] to property (1)

lemma [code abstype]: Dlist (list dxs) = dxs

instructs the code generator to make Dlist a pseudo-constructor and to turn the composition around and make (5) a code equation, although it is not a theorem. The justification is a meta-theoretic one: we ensure that in code equations, Dlist is only applied to distinct lists, for which (5) is provable. This property of Dlist will be guaranteed by a check that Dlist is only applied to the result of operations on lists that have been proved to preserve the invariant. That is, we ensure that the implementations of the dlist operations on lists preserve distinct. For List.remove1, the implementation of Dlist.remove, we need to show

$$\mathsf{distinct}\ xs \Longrightarrow \mathsf{distinct}\ (\mathsf{List.remove1}\ x\ xs) \tag{6}$$

However, we can do better and combine (3) and (5) like this:

lemma list_remove[code abstract]:
  list (Dlist.remove x dxs) = List.remove1 x (list dxs)

The attribute [code abstract] instructs the code generator to derive the actual code equation (3) from it (this is a direct consequence of (1)). The lemma also entails (5): if distinct xs then

$$\begin{aligned}&\mathsf{list}\ (\mathsf{Dlist}\ (\mathsf{List.remove1}\ x\ xs)) = \mathsf{list}\ (\mathsf{Dlist.remove}\ x\ (\mathsf{Dlist}\ xs))\\ &= \mathsf{List.remove1}\ x\ (\mathsf{list}\ (\mathsf{Dlist}\ xs)) = \mathsf{List.remove1}\ x\ xs,\end{aligned}$$

i.e., distinct (List.remove1 x xs) (by (4), list_remove, (2)). Thus lemma list_remove also certifies that distinct is preserved by List.remove1.

This concludes the presentation of code generation for dlist. The advantage of our approach is that we have relaxed the principle to only ever generate code from theorems in only one place, equation (5). Above we sketched why this is admissible. In the next subsection we explain our approach in its general form and give a formal correctness proof.

3.2 Subtype Step: The General Case

Now we look at the general form of the subtype step from 'a list (now C) to 'a dlist (now $A = (\overline{\alpha})\kappa$). We have functions $Abs : C \to A$ (Dlist) and $rep : A \to C$ (list) such that $Abs(rep(y)) = y$ and $inv : C \to bool$ (distinct) such that $inv(x) \longleftrightarrow rep(Abs(x)) = x$. We assume that the result type of all functions in Σ contains κ at most at the very outside, e.g., 'a dlist is allowed but 'a dlist list is not. We discuss this restriction (which does not apply to derived functions) at the end of this section. The format for the code equations is now

$$\psi(f\ \overline{y}) \ = \ t$$

where ψ is rep (if $\tau = (\dots)\kappa$) or the identity (otherwise). The free variables of t must be contained in $\overline{y}$. The code generator turns this into $f\ \overline{y} = \phi(t)$ (by a proof step), where ϕ is Abs (if $\tau = (\dots)\kappa$) or the identity (otherwise). The only liberty that the code generator takes is that it turns the theorem $Abs(rep\ y) = y$ into the non-theorem $rep(Abs\ x) = x$. Of course the latter is implied by $inv(x)$, and we will show that $inv(s)$ holds for all terms $Abs(s)$ that may arise during a computation. But this requires a careful proof (see below). The following table summarizes the behavior of the code generator.

E	E'
$rep(f\ \overline{y}) = t$	$f\ \overline{y} = Abs(t)$
$Abs(rep\ y) = y$	$rep(Abs\ x) = x$

Let E be the set of all code equations at the point when the code generator is invoked and let E' be the result of the translation shown in the table above. That is, most equations are moved from E to E' unchanged, but $rep(f\ \overline{y}) = t$ and $Abs(rep\ y) = y$ are translated as above. Moreover, the code generator enforces that Abs must not occur on the rhs of any equation in E. (This is not a restriction because if one really needed an operation that behaved like *Abs* one could define it separately from *Abs* to avoid confusion.)

In [5] correctness of the code generation process is shown by interpreting code equations as higher-order rewrite rules and proving that code generation preserves the reduction behavior. Our translation from E to E' is a first step that happens before the steps considered in [5]. We will now prove correctness of that first step by relating the equational theory of E (written $E \vdash u = v$) with reduction in E'. Notation $E' \vdash u \to v$ means that there is a rewrite step from u to v using either a rule from E' or β-reduction.

We call a term t *invariant* iff (i) $E \vdash \mathit{rep}(\mathit{Abs}\ s) = s$ for all subterms $(\mathit{Abs}\ s)$ of t and (ii) every occurrence of *Abs* in t is applied to an argument.

Lemma 1. *If u is invariant and $E' \vdash u \to^* v$, then v is invariant.*

Proof. By induction on the length of the reduction sequence. In each step, we need to check invariance of newly created *Abs* terms. Because user-provided code equations with *Abs* on the rhs are forbidden, only the derived code equation $f\ \overline{y} = \mathit{Abs}(t)$ can introduce a new *Abs* term, namely $\mathit{Abs}(t)$ itself, where *Abs* is applied and for which we have $E \vdash \mathit{rep}(\mathit{Abs}(t)) = \mathit{rep}(\mathit{Abs}(\mathit{rep}(f\ \overline{y}))) = \mathit{rep}(f\ \overline{y}) = t$. Invariance is preserved by β-reduction because it cannot create new *Abs* terms because all *Abs* must already be applied to arguments.

Lemma 2. *If u is invariant and $E' \vdash u \to^* v$, then $E \vdash u = v$.*

Proof. By induction on the length of the reduction sequence. In each step, either β-reduction, or an equation from E, or $f\ \overline{y} = \mathit{Abs}(t)$ (which is a consequence of E), or $\mathit{rep}(\mathit{Abs}\ x) = x$ is used. Only the last case needs special consideration. By the previous lemma, the subterm $\mathit{Abs}(t)$ of the lhs of the reduction $\mathit{rep}(\mathit{Abs}(t)) \to t$ is invariant, and hence $E \vdash \mathit{rep}(\mathit{Abs}(t)) = t$.

Thus we know that if we start with an invariant term, reduction with E' only produces equations that are already provable in E. Invariance of the initial term is enforced by Isabelle very easily: the initial term must not contain *Abs*.

We have already mentioned that we cannot register a code equation for a basic operation by [code abstract] if the abstract type κ occurs inside the result type rather than at the top level. A workaround is to introduce for each such result type a new abstract type with appropriate projection functions. For example, $\cdots \to (\alpha)\kappa \times (\alpha)\kappa$ becomes $\cdots \to (\alpha)\kappa'$, where κ' is a new abstract type, a copy of $(\alpha)\kappa \times (\alpha)\kappa$ with two projection functions of type $(\alpha)\kappa' \to (\alpha)\kappa$. This leads to simultaneous refinement, which is covered by our approach. Sometimes the workaround can be avoided because the offending operation can be split up into different functions. For example, a function of type $\cdots \to (\alpha)\kappa \times (\alpha)\kappa$ is replaced by two separate functions of type $\cdots \to (\alpha)\kappa$. We believe that the limitation on the result type can be lifted, but it requires a generalization of the correctness proof by employing map functions for each container type involved.

3.3 Using Lifting/Transfer

Building a theory library that implements a new abstract type like 'a dlist can take a bit of work. The main reason is that the type system requires us to convert between values of the concrete and the abstract type with the isomorphisms. This happens in all definitions, for example of Dlist.remove (3). For more complicated types involving higher-order types or other type constructors, more complex combinations of the isomorphisms are required. Then we need to prove the code equations for [code abstract], e.g., lemma list_remove, from those definitions. And finally we need to transfer properties from the concrete to the abstract type. Thus the manual construction of such an abstract type is at least tedious. If one is unfamiliar with the details of the type definition facility, it is not just tedious but cryptic. Hence the success of our approach to invariants depends on the amount of automation we are able to provide for this task.

To automate the construction of abstract types we use the Lifting and Transfer packages [8], which were implemented as general tools but also with the motivation of data refinement in mind. These tools provide automation for building abstract types (subtypes and quotients) in Isabelle/HOL and were inspired by [10]. The Lifting package defines new constants on the abstract level, which is done by *lifting* terms from the concrete level to the abstract level, and proves *transfer rules* relating a term on the concrete level and the newly defined constant. The Transfer package helps to prove theorems on the abstract level (mainly properties of the lifted constants), which is done by *transferring* the goals on the abstract level to goals on the concrete level by using the provided transfer rules.

How to use Lifting/Transfer for implementation of a data structure with an invariant? First of all, we have to set up the lifting infrastructure, which is done by a theorem generated by **typedef** for 'a dlist:

setup_lifting type_definition_dlist

This canonical boilerplate command already registers 'a dlist as an abstract datatype with a constructor Dlist (via [code abstype]).

Then each operation can be lifted by **lift_definition** command:

lift_definition remove :: 'a ⇒ 'a dlist ⇒ 'a dlist **is** List.remove1

This command opens a proof environment with the following goal:

$\bigwedge a\ list.$ distinct $list \Longrightarrow$ distinct (List.remove1 $a\ list$)

The goal merely expresses that the operation on the concrete level preserves the invariant. After the proof is finished, a new constant remove is automatically defined via the correct combination of isomorphisms and also the corresponding code equation list_remove is proved (from the definition and the above goal) and registered in the code generator via [code abstract].

Transfer helps us to prove properties of operations on 'a dlist, in particular code equations:

lemma [code]: Set.remove x (dset dxs) = dset (Dlist.remove x dxs)
apply transfer

Command transfer turns the goal into

$$\bigwedge x\ dxs.\ \mathsf{distinct}\ dxs \Longrightarrow \mathsf{Set.remove}\ x\ (\mathsf{set}\ dxs) = \mathsf{set}\ (\mathsf{List.remove1}\ x\ dxs),$$

which talks about lists rather than distinct lists and can thus be proved easily. Note that *dxs* is now universally quantified and has type 'a list.

We will now give an abstract description of what the Lifting package does in our setting of data refinement and building subtypes. Let us assume we have for each abstract type $A = (\overline{\alpha})\,\kappa$ two morphisms $Abs_\kappa :: A \to C$, $rep_\kappa :: C \to A$ and an invariant $inv :: A \to \text{bool}$ and we want to lift a function f' to a function f whose type should be τ.

Let us assume that $\tau = \tau_1 \to \cdots \to \tau_{n+1}$, then the Lifting package will ask us to prove the following correctness condition

$$\text{Inv}(\tau_1)\ x_1 \Longrightarrow \ldots \Longrightarrow \text{Inv}(\tau_n)\ x_n \Longrightarrow \text{Inv}(\tau_{n+1})\ (f'\ \overline{x}),$$

where $\text{Inv}(\tau_i) :: \tau_i \to \text{bool}$ is a compound predicate that checks if all values of the concrete type C stored in a "container" of type τ_i meet the invariant. $\text{Inv}(\tau_i)$ is built from the invariant function inv and predicators[1] for corresponding types involved in τ by traversing τ. For example, for $\tau = A =$ 'a dlist, we have $\text{Inv}(\tau) = inv = \mathsf{distinct}$; for $\tau =$ 'a dlist list, we have $\text{Inv}(\tau) = \mathsf{list\_all\ distinct}$. More details are beyond the scope of this paper.[2]

If we prove the correctness condition, the Lifting package will produce the following definition of f

$$f = \text{Def}^{+}(\tau)\ f',$$

where Def is a function that builds a compound morphism by traversing the given type τ. The polarity superscript $+$ (or $-$) encodes if an abstraction (or a representation) function should be generated. $\text{Def}^{p}(\tau)$ is defined by two recursive equations:

- If $\tau = (\overline{\sigma})\,\kappa$ is not an abstract type, then

$$\text{Def}^{p}(\tau) = \text{map}_\kappa\ \text{Def}^{P^1_\kappa(p)}(\sigma_1)\ \ldots\ \text{Def}^{P^n_\kappa(p)}(\sigma_n),$$

  where map_κ is a map function[3] for κ and P^i_κ encodes which type arguments of κ are co-variant or contra-variant: if the i-th type argument is co-variant then P^i_κ is the identity; if contra-variant, P^i_κ yields $-$ for $+$ and $+$ for $-$.
- If $\tau = (\overline{\sigma})\,\kappa$ is an abstract type, then

$$\text{Def}^{p}(\tau) = \text{Morph}^{p}(\kappa) \circ \text{Def}^{p}((\overline{\sigma'})\,\vartheta),$$

[1] A predicator pred_ϑ for $(\alpha)\,\vartheta$ is a function of a type $(\alpha \to \text{bool}) \to (\alpha)\,\vartheta \to \text{bool}$ lifting a predicate inv operating on α's to a predicate $\text{pred}_\vartheta\ inv$ operating on $(\alpha)\,\vartheta$.

[2] The Lifting package is more general than we present here, e.g., the concrete type is also a parameter of lifting; full details will be published in a forthcoming paper.

[3] if κ does not have any type arguments, then $\text{map}_\kappa = \text{id}$

where $(\overline{\alpha})\,\kappa$ is defined as a subtype of $(\overline{\beta})\,\vartheta$ and $\theta = \mathrm{match}(\overline{\sigma}, \overline{\alpha})$ and $\overline{\sigma'} = \theta\ \overline{\beta}$. [4] Function Morph^p gives us the concrete morphism according to the polarity: $\mathrm{Morph}^+(\kappa) = \mathit{Abs}_\kappa$ and $\mathrm{Morph}^-(\kappa) = \mathit{rep}_\kappa$.

In general, if corresponding morphisms, map functions and predicators are known to the system, the Lifting package will produce a correct definition for any combination of abstract and concrete types including higher-order and nested types. The only limitation is that currently the Lifting package does not provide strong enough transfer rules for constants with nested abstract types, which means that the Transfer package would not be able to transfer some goals containing such constants. The current workaround is to provide the stronger transfer rules manually. This limitation is work in progress. A formal description of the Transfer package is the subject of a forthcoming paper.

Except for proving that an operation on the concrete level preserves the invariant, and this proof is in general unavoidable, everything is fully automatic.

4 From Type Constructors to Type Expressions

4.1 Motivation and Example

The limitation of the code generator is that the type that is being refined has to be of the form $(\overline{\alpha})\,\kappa$. The type of *maps* 'a ⇒ 'b option does not have this form, yet one would still like to refine it by some efficient type of tables. Because 'a ⇒ 'b option is not a plain type constructor, a new type ('a, 'b) mapping has to be introduced. This type is merely a copy of 'a ⇒ 'b option for code generation purposes. It can be refined further, for example, by red-black trees using the techniques from §2 and §3. See Figure 3 for the complete picture. The implementation type rbt-impl is just a plain datatype of binary trees with a color in each node; on top of it the subtype rbt of well-shaped trees satisfying the invariant of red-black trees is defined. This example represents the most general form of data refinement discussed in this paper.

But now all definitions using 'a ⇒ 'b option must be lifted to ('a, 'b) mapping. Of course, it has to be done for primitive operations on maps like a lookup or an update only once for all. But it also has to be done for all other definitions using these primitive functions. The reason is that one has to provide for such derived operations new code equations that use primitive operations of ('a, 'b) mapping and not 'a ⇒ 'b option. On the other hand, no code equations have to be provided for the primitive operations on ('a, 'b) mapping in this phase because these will be provided later on in the phase described in §2 and §3. Of course, it is also possible to base the formalization on the lifted type ('a,'b) mapping from the beginning but this contradicts the very idea of data refinement.

The complications of this general setting are as follows. For a start, you do not obtain code for f but for some f'. This means in particular that none of the tools

[4] In our setting it is guaranteed that all type variable in $\overline{\beta}$ are present in $\overline{\alpha}$ and thus $\overline{\sigma}$ uniquely determines $\overline{\sigma'}$.

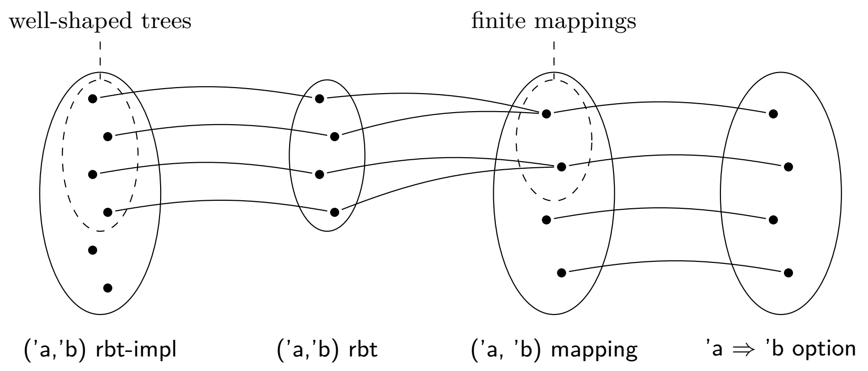

Fig. 3. Maps implemented by red-black trees

that build on code generation, e.g., Quickcheck profit from such refinements. Moreover you have to refine every function f to some f', not just the primitive ones, and you have to look carefully at the definition of f' that **lift_definition** actually produced and at the abstraction relations involved to convince yourself that f and f' are in the desired relationship. But it is not quite as bad as this. As soon as you define a derived function h where 'a ⇒ 'b option is no longer present in the type of h, but which still uses maps inside its body, you no longer need to lift h to some h', but you still have to prove a code equation for h itself that uses mappings internally. This is done again by transfer.

4.2 Using Lifting/Transfer

We can again use the Lifting and Transfer packages to automate the lifting. First, ('a, 'b) mapping is defined as a copy of 'a ⇒ 'b option and all primitive operations on maps are lifted:

```
typedef ('a, 'b) mapping = UNIV :: ('a ⇒ 'b option) set ..
setup_lifting(no_code) type_definition_mapping

lift_definition empty :: ('a, 'b) mapping is (λ_. None) .
lift_definition lookup :: ('a, 'b) mapping ⇒ 'a ⇒ 'b option is λm k. m k .
lift_definition update :: 'a ⇒ 'b ⇒ ('a, 'b) mapping ⇒ ('a, 'b) mapping
  is λk v m. m(k ↦ v) .
```

We showed only 3 such operations here but in reality there are more of them. Notice that we do not have to prove anything in the **lift_definition** command because the formal invariant preservation theorem is proved automatically if we work with type copies.

Now let us assume we used maps in our formalization to implement a special data type that behaves like a multiset and the multiplicity of elements is limited.

Now we can implement an insert for this data structure that ensures that if the limit is reached, the map is not changed.

```
definition insert_lim :: ('a ⇒ nat option) ⇒ 'a ⇒ nat ⇒ 'a ⇒ nat option
where insert_lim m k lim = (case m k of
  Some n ⇒  if n < lim then m(k ↦ n + 1) else m
  | None ⇒ m(k ↦ 1))
```

We use **lift_definition** to define a copy of insert_lim that operates on the code generation type ('a, nat) mapping.

```
lift_definition insert_lim′ :: ('a, nat) mapping ⇒ 'a ⇒ nat ⇒ ('a, nat) mapping
  is insert_lim .
```

Function insert_lim′ is defined in terms of the original function insert_lim with the help of the morphisms between maps and mappings. In contrast to the situation in §3, we cannot use this definition as a code equation because it goes in the wrong direction: it reduces a computation on mappings to maps. The desired code equation for insert_lim′ is proved by transfer from the definition of the original function.

```
lemma [code]: insert_lim′ m k lim = (case Mapping.lookup m k of
  Some n ⇒ if n < lim then Mapping.update k (n + 1) m else m
  | None ⇒ Mapping.update k 1 m)
by transfer (fact insert_lim_def)
```

It is inconvenient that one has to write down the lifted code equation even if the proof is trivial thanks to Lifting/Transfer. In principle we can use the Transfer package to transfer goals in the other direction, i.e., from the concrete level to the abstract level and thus we would not have to write down the lifted code equation at all. But there is the problem that if we go in this direction, it is not clear which parts of a term should really be transferred. The transfer method can eagerly transfer all terms from the 'a ⇒ 'b option to the ('a, 'b) mapping level according to the transfer rules. But maybe the user would want some subterms to remain maps. This would require some mechanism that allows users to annotate a term and say which parts should not be transferred. This is work in progress and we intend to profit from the heuristics developed by Lammich [11]. Transferring from ('a, 'b) mapping to 'a ⇒ 'b option instead is unambiguous: all occurrences of mapping are replaced.

5 Applications

The following examples are the most important applications of data refinement in the Isabelle distribution.

Sets are implemented by lists by default. There is also an efficient implementation by red-black trees (in `Library/RBT_Set.thy`). In a recent application [18] a decision algorithm for MSO formulas was unusable with the default implementation of sets, but when theory `RBT_Set.thy` was loaded (no change of the client code is necessary!), it allowed us to decide small MSO formulas.

Mappings were described in §4. The distribution provides two implementations: red-black trees (as in §4) and association lists $(\mathsf{'a} \times \mathsf{'b})$ list.

Integers (type int) are defined as a total quotient of pairs of natural numbers $\mathsf{nat} \times \mathsf{nat}$ by the Lifting and Transfer packages. Two pairs of natural numbers (x, y) and (u, v) represent the same integer if $x + v = u + y$. We do not use this definition for execution of integers because of efficiency reasons but we execute them by binary numerals num defined as a datatype:

$\textbf{datatype}$ $\mathsf{num} = \mathsf{One} \mid \mathsf{Bit0}\ \mathsf{num} \mid \mathsf{Bit1}\ \mathsf{num}$

Operations on num are just usual binary arithmetic. Then, all integers are interpreted as binary numerals by employing three pseudo-constructors $\mathsf{0} :: \mathsf{int}$, $\mathsf{Pos} :: \mathsf{num} \Rightarrow \mathsf{int}$ and $\mathsf{Neg} :: \mathsf{num} \Rightarrow \mathsf{int}$. The last step is to implement common integer operations by pattern-matching on these three pseudo-constructors and using corresponding operations on num.

Rationals (type rat) are defined as a partial quotient of pairs of integers $\mathsf{int} \times \mathsf{int}$, again with the help of Lifting and Transfer. The quotient is partial because we do not include pairs $(\_,0)$ with a zero denominator. This is a logical definition of rational numbers used for formalizations but because the quotient is partial, we cannot use it directly for execution. Instead we interpret type rat as a subtype of $\mathsf{int} \times \mathsf{int}$ based on an observation that each rational number can be represented by a pair of co-prime integers with a non-zero denominator. Given the pseudo-constructor $\mathsf{Frct} :: \mathsf{int} \times \mathsf{int} \Rightarrow \mathsf{rat}$, the *rep* function $\mathsf{quotient\_of} :: \mathsf{rat} \Rightarrow \mathsf{int} \times \mathsf{int}$ is defined as follows

$\mathsf{quotient\_of}\ r = (\mathsf{THE}\ (n,\ d).\ r = \mathsf{Frct}\ (n,\ d) \wedge d > \mathsf{0} \wedge \mathsf{coprime}\ n\ d)$

and allows us to use the invariant mechanism described in §3 and execute rational numbers.

Reals (type real) are executed by rationals using the pseudo-constructor $\mathsf{Ratreal} :: \mathsf{rat} \Rightarrow \mathsf{real}$ and code equations for $+$, $-$, $*$, $/$, but not much more because only the rational reals are representable in this manner. But it is still useful. For example, it enables Quickcheck to find rational counterexamples to conjectures involving polynomials.

Basic arithmetic on complex numbers is executable without data refinement.

Outside the Isabelle distribution, data refinement has found a number of applications, too. For example, five entries in the *Archive of Formal Proofs* `http://afp.sf.org` define their own data abstractions, some of which are also discussed in the literature [13].

6 Related work

Data refinement is a perennial topic that was first considered by Hoare more than 40 years ago [7], who already introduced abstraction functions and invariants. This principle of data refinement became an integral part of the model oriented specification language VDM [9] (and was later generalized to nondeterministic operations [14,6]). In the first-order context of universal algebra it was shown that there are always fully abstract models such that any concrete implementation can be shown correct with a homomorphism [15].

The infrastructure presented in this paper has been available in Isabelle for a few years but has never been published properly: [5] merely shows an example (similar to §2); the core of the present paper, the treatment of invariants as in §3, was not available at that time. Nevertheless the infrastructure has already been used in many places (see §5). Based on this infrastructure, Lochbihler [12] has recently overcome some limitations of our approach (e.g., see the end of §2).

Lammich [11] has implemented a new framework for data refinement that has some similarity with §4: you do not obtain code for f but for some f' that is in a certain relationship to f. As a result he can work with a more general notion of refinement supporting (for example) nondeterministic operations and multiple implementations of the same type. Of course he also faces the complications explained in §4.1. A difference is that his system proves invariance preservation for derived functions as an explicit theorem whereas for us the type checker does the work. In a nutshell, his is a general framework for heavy duty data refinement, ours is a lightweight infrastructure for completely transparent but more limited data refinement.

ACL2 also seems to provide for data refinement based on invariants [4], but the exact relationship is unclear. In Coq [1], parametrized modules support a form of data refinement [3]: perform your development inside the context of a specification of finite sets (or whatever abstract type you have), and later instantiate the module with some implementation of finite sets that has been proved to satisfy the finite set axioms. The drawback is that you do not really work with the actual abstract type (e.g., sets), but some axiomatization of it, which may not have the same nice syntax and proof support.

7 Conclusion

We have presented Isabelle/HOL's infrastructure for a lightweight approach to data refinement. Its distinctive feature is the tight integration with the code generator and hence also any tool that builds on it. The key principle is that when you want to execute function f, you really execute f, which in turn calls a more efficient implementation f' that was proved equivalent to f. As a result, data refinement is completely transparent to the user: just load a specific refinement theory and the code generator does the rest. This completely automatic approach assumes that refinement happens on a per type constructor basis. To remove this assumption we presented a more general approach that sacrifices some of the above advantages. It relies strongly on two packages for lifting definitions and theorems from one type to another automatically.

References

1. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

2. Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic Proof and Disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, FroCoS 2011*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.
3. Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *European Symposium on Programming, ESOP 2004*, volume 2986 of *LNCS*, pages 370–384. Springer, 2004.
4. D. Greve, M. Kaufmann, P. Manolios, J Moore, S. Ray, J. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient execution in an automated reasoning environment. *J. Functional Programming*, 18:15–46, 2008.
5. Florian Haftmann and Tobias Nipkow. Code Generation via Higher-Order Rewrite Systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *FLOPS*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.
6. J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *ESOP '86: European Symposium on Programming*, volume 213 of *LNCS*, pages 187–196. Springer, 1986.
7. C.A.R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.
8. Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. Presented at the Isabelle Users Workshop at ITP 2012. `http://www21.in.tum.de/~kuncar/huffman-kuncar-itp2012.pdf`, 2012.
9. Cliff B. Jones. *Software Development. A Rigourous Approach*. Prentice Hall, 1980.
10. Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proc. of the 26th ACM Symposium on Applied Computing (SAC'11)*, pages 1639–1644. ACM, 2011.
11. Peter Lammich. Automatic Data Refinement. Submitted to ITP 2013, 2013.
12. Andreas Lochbihler. Light-weight containers for Isabelle: efficient, extensible and nestable. Submitted to ITP 2013, 2013.
13. Andreas Lochbihler and Lukas Bulwahn. Animating the Formalised Semantics of a Java-like Language. In Van Eekelen, Geuvers, Schmaltz, and Wiedijk, editors, *Interactive Theorem Proving, ITP 2011*, volume 6898 of *LNCS*, pages 216–232. Springer, 2011.
14. Tobias Nipkow. Non-Deterministic Data Types: Models and Implementations. *Acta Informatica*, 22:629–661, 1986.
15. Tobias Nipkow. Are Homomorphisms Sufficient for Behavioural Implementations of Deterministic and Nondeterministic Data Types? In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Symp. Theoretical Aspects of Computer Science, STACS 87*, volume 247 of *LNCS*, pages 260–271. Springer, 1987.
16. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
17. Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95: Proc. Tenth Annual Conf. Computer Assurance*, pages 151–162. IEEE, 1995.
18. Dmitriy Traytel and Tobias Nipkow. A Verified Decision Procedure for MSO on Words. `http://www.in.tum.de/~nipkow/pubs`, 2013.