Simplifying Automated Data Refinement via Quotients

Alexander Krauss

July 2011

Abstract

This report gives an overview over problems related to the automatic transport of theorems from one type to another. Starting from the original motivation, data refinement during code generation, we explore the possibilities and the existing implementations of related mechanisms in Isabelle. In particular, we show how formalizations over ordinary sets can be automatically transported over to the isomorphic but executable copy α *Set* (commonly known as *Cset.set*). This approach could be a cornerstone in a general and practical approach to data refinement.

Moreover, we give some comparison of the existing tools *transfer* (by Chaieb and Avigad) and the quotient package (by Kaliszyk and Urban), exploring the question whether these tools could be unified. This document does not present finished solutions in this regard. Rather, its purpose is to give a better understanding of the problems and bring relevant information together that has been disconnected, hard to find, or undocumented.

1 Motivation: Data Refinement

Isabelle's code generator can produce executable code from HOL specifications that have a certain form resembling functional programs. Such specifications are called *executable*.

The semantics of code generation is simple: Any reduction $t \rightarrow v$ in the generated program is also valid as an equation $t = v$ in HOL (modulo the translation). In other words, the generated program is a sound rewriting engine with respect to the original theory. There are no other guarantees such as termination or any form of completeness.

1.1 Code Generation Scenarios

From a user's perspective, there are at least four different ways of coming into contact with code generation. We will refer to them as *code generation scenarios*:

Program verification The user writes an executable specification and proves some properties about it. Then, the code generator is invoked explicitly to write code to a file, which is then used externally, with the knowledge that it has been verified (although the verification itself is not exported).

Ad-hoc testing The user invokes the *value* command to evaluate some expressions for testing purposes. This invokes the code generator, together with a reverse translation for the result.

Random testing The user invokes the *quickcheck* command to find counterexamples for a lemma. This invokes the code generator on the lemma, with some random value generators (written in HOL) fused in.

Proof by evaluation The user proves an equation $t = v$ by evaluating t to v, or a statement by evaluating it to *True*. This is done using a special *evaluation* oracle (and corresponding method), which trusts the correctness of the code generation and execution.

1.2 What is Data Refinement?

When developing theories, it is usually desirable to express algorithms abstractly in terms of functions, sets, multisets etc. On the other hand, we expect these specifications to be executed efficently using data structures like lists or balanced trees, depending on the application. This step from abstract to concrete data structures is called *data refinement.*

Unfortunately, data refinement is harder to accomplish than algorithm refinement, i.e., replacing one algorithm with a more efficient one. The latter is easy, because HOL views equivalent algorithms as the same function (by extensionality), so we can derive, e.g., the equation $\mathit{isort} = \mathit{quicksort}$ and then use the slow but simple function for proofs and the fast but complicated function for execution.

For data structures this is not so easy: In HOL, two types cannot be "extensionally equal" just like two functions that compute the same result. In the best case, they are isomorphic, but we will see below that this is almost never the case. Moreover, replacing one type with another also requires replacing the relevant operations, in such a way that the resulting statement is type correct (and valid). In particular, specifications are full of function types, but only some of them are supposed to become balanced trees in the generated code.

1.3 Example: Sets by Distinct Lists

The following example demonstrates all the issues that may arise in the context of data refinement. Consider the type of sets (represented by $\alpha \Rightarrow \mathit{bool}$ in Isabelle), which we want to implement by lists. Let us assume we do not care about the order of the elements in the list, but we do not want elements to occur repeatedly. (We call such lists as *distinct lists* below.) This is not extremely efficient, but it is simple, does not require an order on the elements, and prevents performance degradation when the same element is added to a set many times.

Clearly, sets and lists are not isomorphic structures, but the situation is quite a bit more complicated and depicted in Fig. 1: First, there are *unrepresentable elements*: clearly, an infinite set cannot be implemented by a list. Second, there are *illegal representations*: the list $[0, 0]$ violates our invariant and thus does not represent a set (although we could give a meaningful interpretation of such lists, this need not be the case in general). Finally, an abstract value may have *multiple representations*: the set $\{0, 1\}$ can be expressed by the lists $[0, 1]$ and $[1, 0]$, and there is no preference for one of them. In other words, between the set of representable elements, and the set of legal representations, we have a quotient.

In addition, it is a technical restriction of the code generator that data refinement can only be applied to type constructors $\alpha\ \kappa$, not arbitrary "open types" like $\alpha \Rightarrow \mathit{bool}$. This means that before we can successfully apply the techniques below, we must first introduce an explicit type constructor $\alpha\ \mathit{Set}$, which is isomorphic to $\alpha \Rightarrow \mathit{bool}$. It is a somewhat unfortunate coincidence that this explicit type constructor was abolished in Isabelle2008 and we must now introduce it again for code generation. However, even if sets were an explicit type constructor right from the start, the same problem would apply for maps (of type $\alpha \Rightarrow \beta\ \mathit{option}$).

To address the data refinement problem situation, Haftmann [2] proposes a stepwise approach, with several layers between sets and lists, each of which is closer to the final implementation. Figure 2 shows the different layers (where we added the extra layer "Implementable subtype", which is not actually present in the implementation but helps to understand the nature of the setting).

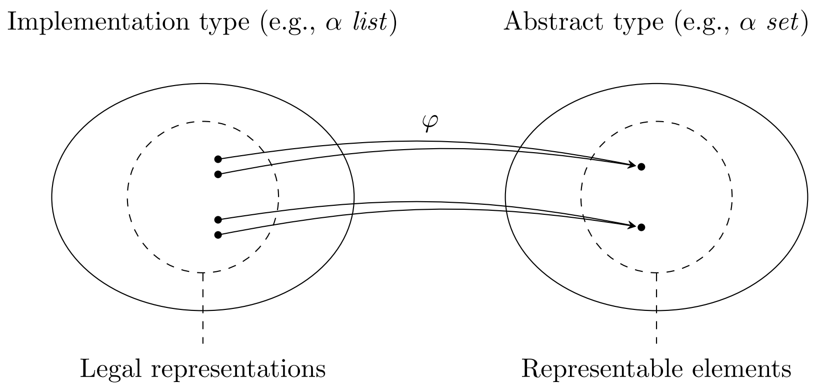
Theory Reference. Theory `HOL/Library/Cset.thy`[1] defines a copy of sets with an explicit type constructor *Cset.set* (which we will continue to call *Set* for brevity). Likewise, a copy of maps of type $\alpha \Rightarrow \beta\ \mathit{option}$, which is called $(\alpha, \beta)\ \mathit{mapping}$, can be found in `HOL/Library/Mapping.thy`.

In the following, we describe the data refinement setup from the explicit type constructor downwards. For these steps, satisfactory solutions exist and are already implemented. Thus, by importing the appropriate theories, a user directly obtains executable code for type $\alpha\ \mathit{Set}$ without further configuration. The final gap—the connection between the abstract type and the explicit type constructor—will concern us later.

1.4 Haftmann's Trick

The connection between $\alpha\ \mathit{Set}$ to $\alpha\ \mathit{dlist}$ (i.e., the treatment of both unrepresentable elements and multiple representations) uses what should best be called *Haftmann's Trick*, as it allows the replacement of one type by another one with

[1]All theory references refer to `Isabelle 9959c8732edf`.



Implementation type (e.g., α *list*)
Abstract type (e.g., α *set*)
φ
Legal representations
Representable elements

Figure 1: Relationship between implementation type and abstract type

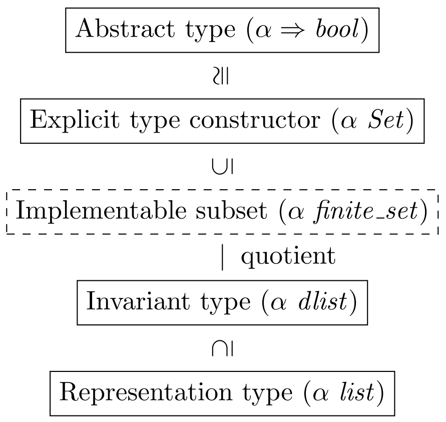

Abstract type ($\alpha \Rightarrow bool$)
Explicit type constructor (α *Set*)
Implementable subset (α *finite_set*)
quotient
Invariant type (α *dlist*)
Representation type (α *list*)

Figure 2: Intermediate layers used in code generation

surprising ease, based purely on the equational semantics of code generation outlined above. It was first used in Haftmann's thesis [1, ch. 4.1] to refine naive queues to amortized queues, and has found other nice applications in the meantime (e.g., [5]).

The basic idea is that due to the purely equational view of the generated code, a datatype is nothing more than a collection of uninterpreted functions—its constructors. So we are free to pick any constants (of the appropriate types) and turn them into datatype constructors in the generated code. We call such constants pseudo-constructors. Like ordinary constructors, they do not carry any code equations, but other code equations may instead use them in patterns on the left hand side. There are no particular properties that such pseudo constructors must satisfy, i.e., they need not be injective or exhaust the type.

Even without these properties, it is easy to convince ourselves that all this is just fine under our equational interpretation, since all reduction steps in the program still arise from valid theorems.

In our concrete case, we define a function $Set :: \alpha\ dlist \Rightarrow \alpha\ Set$, and declare it as a pseudo-constructor. Thus, in the generated code, the type α *Set* will become a datatype, whose elements are in fact *dlist*s. We now derive the code equations that treat the function *Set* as a constructor. E.g., for set union we have

$$Set\ A \cup Set\ B = Set\ (Dlist.union\ A\ B)$$

where *Dlist.union* implements the union on *dlist*. These equations are easy to derive.

Theory Reference. A setup of the implementation of α *Set* by α *dlist* as described here can be found in `HOL/Library/Dlist_Cset.thy`. Similarly, `HOL/Library/RBT_Mapping.thy` relates (α, β) *mapping* to red-black trees.

Such setups are tedious to produce, since operations are duplicated on the different levels and must be related explicitly via lemmas. However, this must be done only once for each data structure and can therefore be hidden in a collections library. A user of the data structure can then use the abstract view only, and gets an efficient implementation for free.

1.5 Invariants

The transition from distinct lists down to plain lists is a little more subtle, semantically, and unlike Haftmann's Trick above, it requires an extension of the code generator and its soundness proof, which considers the invariants.

Basically, we must make sure that the functions in the generated code always preserve the invariant, and this is done by requiring the code equations to have a certain specific form.

We assume that the invariant type (here, α *dlist*) and the implementing type (here α *list*) are related via two functions $\mathit{Abs} :: \alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{dlist}$ and $\mathit{Rep} :: \alpha\ \mathit{dlist} \Rightarrow \alpha\ \mathit{list}$, such that $\mathit{Abs}\ (\mathit{Rep}\ x) = x$ holds unconditionally. We say that $x :: \alpha\ \mathit{list}$ is *invariant* iff it satisfies $\mathit{Rep}\ (\mathit{Abs}\ x) = x$. This situation arises naturally, when the invariant type is defined via *typedef*.

Now, the code generator accepts special code equations that deviate from the normal format. Union would be specified by the equation

$$\mathit{Rep}\ (\mathit{Dlist.union}\ \mathit{xs}\ \mathit{ys}) = \mathit{List.union}\ (\mathit{Rep}\ \mathit{xs})\ (\mathit{Rep}\ \mathit{ys})\ .$$

Note that this equation also encodes that *List.union* preserves the invariant. In the generated code, this equation is turned into a legal function definition in the target language.

```
Dlist.union xs ys = Abs (List.union (Rep xs) (Rep ys))
```

To prove that this transformation is legal, we must show (on paper, by induction on the length of a rewrite sequence), that the invariant is preserved in every rewrite step. However, we do not go into details here.

Theory Reference. A setup of the implementation of distinct lists by normal lists as described here can be found in `HOL/Library/Dlist.thy`. Theory `HOL/Library/RBT.thy` defines a type of red-black trees (with invariant), and relates it to an implementation of raw trees, defined in `RBT_Impl.thy`. Again, these setups are tedious, but end users who merely use the data structures in their theories do not come in contact with them.

1.6 The Missing Link

The remaining issue is that users of data refinement must still formulate their programs in terms of the nonstandard types α *Set* and (α, β) *mapping*. Reasoning support for these types is not very good, since they were introduced only for code generation. Even if tool support were better, replacing normal sets/maps by "code generation sets/maps" still contradicts the very idea of data refinement.

An automatic way of transporting theorems and definitions from one type to another isomorphic one would be a nice solution to this problem. Then, one could proceed as follows in the program verification scenario:

- The library provides a finished setup of the type α *Set*, the setup for code generation as described above, and the transport machinery from α *set* to α *Set*.
- The user writes his programs and proofs using normal sets.

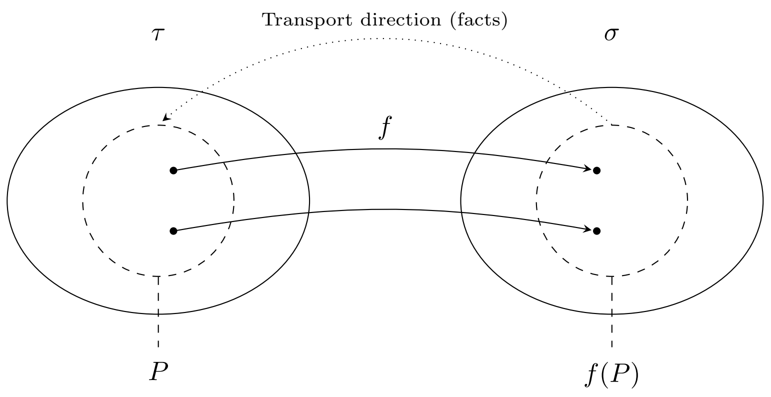


Figure 3: The *transfer* setting

- The user defined constants and their code equations are transported to the executable types. If desired, the properties can also be transported.

In the rest of this report, we investigate existing tool that could help with this task.

Note on terminology. Throughout this report, we use ther term *transport* to refer to the generic idea of moving results from one type to another. This is mainly becauso the terms *transfer* and *lifting* are already used to refer to specific mechanisms.

2 The Transfer Tool

This section describes the *transfer* tool developed by Chaieb and Avigad, which is available in the Isabelle distribution. Originally, *transfer* was intended to ease the transition between integers and natural numbers, and despite the generality of the approach, this remains the only application of the tool up to now, and its only documentation.

The main virtue of *transfer* is its remarkable simplicity. It is based on minimal assumptions, and the implementation is small. However, this also means that it lacks some desirable features, as we will see below. The tool is not limited to isomorphisms between types and can in principle transport theorems from some type σ to a subset of some other type τ. This allows us to see the positive integers as isomorphic to the natural numbers.

A *transfer morphism* is given by types σ and τ, a function $f :: \tau \Rightarrow \sigma$, and a predicate $P :: \tau \Rightarrow bool$. Notably, there are no constraints whatsoever on f and P, so any function can serve as a morphism, bijective or not. We do not specify in advance which operations belong to the types in question. This is done implicitly via a simpset and under user control. However, in the intended use, the function f is injective (we have an isomorphism between the subsets). Figure 3 shows the setting. Note that the transport direction is “backwards”, compared to the morphism f.

Example 1. Between types *int* and *nat* there are two transfer morphisms. The first is given by the function $nat :: int \Rightarrow nat$ and the predicate $(\lambda x.\, x \geq 0)$. The second is given by the function $int :: nat \Rightarrow int$ and the predicate $(\lambda x.\, True)$.

Each transfer morphism is associated with a simpset, which contains rewrite rules describing the interaction between the morphism and the operations on the types (such as addition and multiplication on *nat* and *int*).

When given a theorem, *transfer* proceeds as follows:

1. Each schematic variable $?x :: \sigma$ is instantiated to $f\ (y :: \tau)$, where y is a fresh variable, which will become schematic in a final export operation. Moreover, we add the assumption $P\ y$ for each new variable.

2. The theorem is simplified using the associated simpset. The rules in the set are oriented such that they propagate the morphism upwards, i.e., we have rules of the following form:

$$\begin{aligned} int\ x + int\ y &= int\ (x + y) \\ int\ x * int\ y &= int\ (x * y) \\ (0 :: nat) &= int\ 0 \\ x \geq 0 \implies y \geq 0 \implies nat\ x + nat\ y &= nat\ (x + y) \\ x \geq 0 \implies y \geq 0 \implies nat\ x * nat\ y &= nat\ (x * y) \\ (0 :: int) &= nat\ 0 \end{aligned}$$

   Note that the simplifier will discharge the conditions using the extra assumptions that were added in the previous step. In the end, the morphism has propagated up the term and will eventually disappear by means of rules of the form

$$\begin{aligned} int\ x = int\ y &\longleftrightarrow x = y \\ x \geq 0 \implies y \geq 0 \implies nat\ x = nat\ y &\longleftrightarrow x = y \end{aligned}$$

   which are in the same simpset. This is the point where injectivity of the morphism comes into play,[2] even if it was not required for the declaration of the morphism. Similarly, relations like $\leq$ are replaced by the corresponding relation on the other type.

Example 2. Consider the simple unit law $(?x :: nat) + 0 = x$. After instantiating and adding the assumption, we obtain $(y :: int) \geq 0 \implies nat\ y + (0 :: nat) = nat\ y$. Then the following rewrite sequence produces the final theorem:

$$\begin{aligned} & (y :: int) \geq 0 \implies nat\ y + (0 :: nat) = nat\ y \\ \equiv\ & (y :: int) \geq 0 \implies nat\ y + nat\ 0 = nat\ y \\ \equiv\ & (y :: int) \geq 0 \implies nat\ (y + 0) = nat\ y \qquad (*) \\ \equiv\ & (y :: int) \geq 0 \implies y + 0 = y \end{aligned}$$

Note that step $(*)$ uses a conditional rewrite rule, whose premises must be discharged recursively.

[2]At least this seems to be the intended usage. It may be possible to go without injectivity and replace equality by an equivalence relation instead

2.1 Usage of Transfer

To use *transfer*, one first has to declare a transfer morphism by means of a vacuous theorem carrying a special attribute. Rules can then be added to the respective simpset using further attributes. Transporting actual theorems is done via yet another attribute.

It is not possible to automatically tranfer definitions. Moreover, the tool works as a forward transformation only and cannot be applied backwards on a goal. E.g., given a goal over natural numbers, it is not possible to convert it into a goal over integers.

The tool uses some heuristics for guessing the transfer morphism to be used from the given theorem. We neglect these user interface issues here.

Theory Reference. The only real usage of *transfer* in the system is currently the connection between naturals and positive integers, whose setup can be found in `HOL/Nat_Transfer.thy`. The number theory development uses this connection extensively. The same connection is demonstrated by some toy examples in `HOL/ex/Transfer_Ex.thy`.

2.2 Limitations of Transfer

The desire to keep the *transfer* tool simple has led to some design decisions that may have to be revised when missing functionality is added.

First, the approach is essentially first order. The tool is able to replace *nat* by *int*, but cannot replace *nat list* by *int list*, or *nat* ⇒ *bool* by *int* ⇒ *bool*. For these generalizations, one must be able to construct isomorphims of higher types, which is outlined in the next section. Note that some modest higher-orderness can be obtained by adding certain congruence rules to the simpset, but it is somewhat unclear how far this carries beyond the current usage patterns.

Second, the tool cannot operate on goals. Such a backwards mode of operation could be added in two ways:

1. Replace plain instantiation by a rewrite step of the form $(\bigwedge(x :: \sigma).\, Q\ x) \equiv (\bigwedge(y :: \tau).\, P\ y \implies Q\ (f\ y)$. This makes the approach less general, since it now requires that f is surjective.

2. Generate the new goal externally, assume it as a theorem and apply the forward transformation in the different direction. Then compose the resulting implication with the original goal. This does not demand surjectivity, since we are using the reverse direction. However, one must compute the new goal in advance, which is not possible in the current implementation, which relies entirely on forward simplification.

Third, the desire to map definitions automatically does not go along with the view that the user must manually set up the appropriate simplification rules. More guidance by the system is required here.

Finally, the tool currently does not work with parameterized types, which is apparently due to a bug in the handling of contexts and types. This can presumably be fixed.

3 Aggregate Isomorphisms

This section sketches how isomorphisms between two types σ and τ (we ignore paramenterized types for now) give rise to isomorphisms between higher types that contain σ and τ. This seems to be a standard construction (reference???), and is necessary to build a transport machinery that is not limited to a first-order fragment. The quotient package, discussed below, uses a similar approach to build aggregate equivalence relations over higher types, and we show the simpler case of isomorphisms here mainly to give some basic understanding of the ideas behind it.

Note that the setting in the last section was more general, as we actually considered a subset of type τ. Here, we merely show the simplest setting, where two types σ and τ are related via total bijections $\phi_{\sigma,\tau} :: \sigma \Rightarrow \tau$ and $\psi_{\sigma,\tau} :: \tau \Rightarrow \sigma$ that are inverses of each other.

The morphisms ϕ and ψ can be lifted to more complex type expressions in a canonical way. We call the resulting functions *aggregate isomorphisms.*

$$
\begin{aligned}
\phi_{\sigma_1 \Rightarrow \sigma_2, \tau_1 \Rightarrow \tau_2} &= \lambda f. \psi_{\tau_1,\tau_2} \circ f \circ \phi_{\sigma_1,\sigma_2} \\
\psi_{\sigma_1 \Rightarrow \sigma_2, \tau_1 \Rightarrow \tau_2} &= \lambda f. \phi_{\tau_1,\tau_2} \circ f \circ \psi_{\sigma_1,\sigma_2}
\end{aligned}
$$

$$
\begin{aligned}
\phi_{\sigma_1\ \kappa, \tau_1\ \kappa} &= map_\kappa\ \phi_{\sigma_1,\tau_1} \\
\psi_{\sigma_1\ \kappa, \tau_1\ \kappa} &= map_\kappa\ \psi_{\sigma_1,\tau_1}
\end{aligned}
$$

For type constructors κ, the function map_κ denotes the corresponding map function, e.g. *List.map* for lists. Such map functions exist for all datatypes, and could in principle be generated automatically.

The justification for actually calling these bijections 'isomorphisms' is that we can easily add structure to the types in the form of operations. Given a constant c of some type involving σ and a corresponding constant c' of some type involving τ, we can express their relationship as $c' = \phi\ c$ for the appropriate aggregate isomorphism ϕ. If we do not already have c' we can define it like this.

For example, consider operations $f : \sigma \Rightarrow \sigma$ and $f' : \tau \Rightarrow \tau$. The equation $f' = \phi_{\sigma \Rightarrow \sigma, \tau \Rightarrow \tau}\ f$ unfolds to (using extensionality and the definition of the aggregate isomorphism)

$$\forall x.\ f'\ x = \phi_{\sigma,\tau}\ (f\ (\psi_{\sigma,\tau}\ x))\ ,$$

and applying ψ on both sides yields the familiar

$$\forall x.\ \psi_{\sigma,\tau}\ (f'\ x) = f\ (\psi_{\sigma,\tau}\ x)\ .$$

Note that when defining f' from f via the aggregate isomorphism, we do not care what the original definition of f was, and we do not have to create an "analogous" definition, which could be very complicated. Instead, f' can be defined in terms of f, and the definition of f can later be transported just like any other theorem.

4 The Quotient Package

The quotient package [4] also includes automation for transporting definitions and theorems (which is called 'lifting' in that context). It can automatically

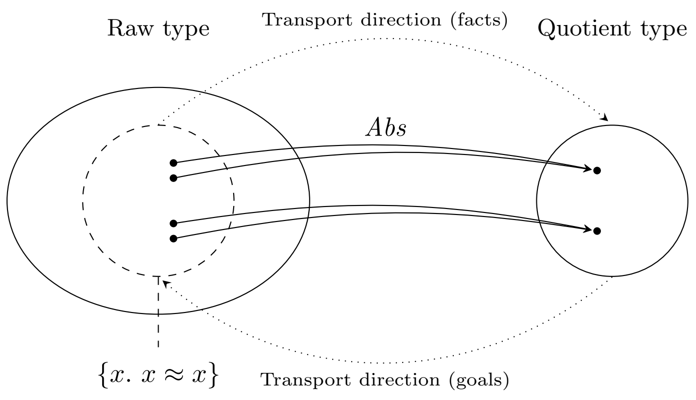


Figure 4: The setting of the quotient package

produce definitions and theorems on the quotient type, given the corresponding constants and facts on the raw type.

The current implementation assumes that the quotient type was actually defined via the same package, but generalizing the code to also work for types defined by other means seems quite easy. However, unlike the *transfer* tool, it is assumed that the morphism from the raw type to the quotient type is surjective, i.e., each element in the quotient type represents an equivalence class of some relation $\approx$. Note that the package works with partial equivalence relations (although this view is not completely exposed to users), which arise naturally since equivalence relations give rise to partial equivalence relations on the function type. Unlike total equivalence relations, partial equivalence relations are not reflexive, and we call the set $\{x.\ x \approx x\}$ the *domain* of the relation. Only values in the domain correspond to elements in the quotient type, an thus the domain has the same role as the predicate P in the *transfer* setting. The morphism that maps from the raw type to the quotient type is called *Abs*, in analogy to *typedef.* Its inverse *Rep* is defined automatically using Hilbert's choice operator. The situation is shown in Fig. 4.

When we ignore and oversimplify many fine points, the lifting procedure works roughly as follows. We assume that the theorem is a closed HOL proposition, i.e., no meta-level constructs and no free variables.

- The raw theorem is "regularized", to bring it into a certain liftable form.
- All constants c are replaced *Rep* (*Abs* c), for the aggregate *Rep* and *Abs* functions of the appropriate type. Note that this is not simply a rewrite step, since the corresponding equation does not hold in general. However, the equivalence $c \approx$ *Rep* (*Abs* c) does hold, and thus we obtain that the original proposition is $\approx$-equivalent to the new one, always using the relation on the appropriate type. On the outer level, this is just equality, so the propositions are logically equivalent.
- A final cleaning step simplifies the proposition by unfolding the definitions of the aggregate morphisms. This creates terms of the form *Abs* (*Rep* t)

at every application position, which simplify to t, this time using plain rewriting.

Note that in contrast to *transfer*, there is no special treatment of outermost schematic variables. In the fully internalized form, all variables are bound (and in fact, the regularization step replaces the quantifiers with bounded ones). As a consequence, bound variables can be treated naturally.

In general, we must know the statement of the theorem that we want to obtain via lifting, since there are many possibly results in general. However, there exist heuristics for guessing a suitable proposition.

4.1 Quotient vs. Transfer

Like *transfer*, the quotient package covers Isomorphisms as a degenerate case, where the equivalence relation $\approx$ is equality. Currently, this requires that the type to which we want to transport theorems was defined as a quotient. Then we obtain very advanced support, which includes lifting of definitions, a lifting of goals that can be used as a tactic, and a good support of higher-order situations.

One limitation is that the *Abs* function must be surjective. This is always the case when the type is actually a quotient. But it prevents the use of the lifting machinery to map theorems from *nat* to *int* (seeing *int* as a superset of a trivial quotient over *nat*).

It is interesting to note that the transport directions of *transfer* and *quotient* are exactly the opposite. While *transfer* encourages the declaration of the inverse direction as a separate transfer morphism, this is usually not possible for quotients.

4.2 Current Technical Limitations

- Currently, the quotient relation must be a dedicated constant. This is inconvenient when we want to use equality, since it requires an extra constant which must be unfolded frequently (but making the definition a `[simp]` rule helps in practice).
- Using `quotient_definition`, one can only lift constants, not arbitrary terms. This prevents the use of the tool on things like locale parameters and some definitions in a local theory.
- One cannot turn a type defined by other means into a quotient afterwards. Similarly, one cannot declare a user-defined constant on the quotient type as the lifted version of another constant.

5 Closing the Code Generation Gap with Quotients

This section shows how to use the quotient package to close the gap outlined in §1.6. Recall that we were using a copy of the abstract set type. After the discussion above, it should be obvious that we should define this type as a quotient.

No manual definitions are made for operations on the new type; instead, we move over all relevant operations from the original type. This works as expected, and is much more canonical than the current setup, which involves a combination of manual transport and reproving. There is no separate reasoning setup. The preferred way of proving a theorem on the new type is by transporting a theorem from the old type, which can be done conveniently at any time.

When a specific implementation of sets is chosen (e.g. distinct lists), we must formulate the corresponding code equations, which are proved on the set level and then transported over.

Theory Reference. An experimental quotient version of `Cset.thy` is now available in the directory `HOL/Quotient_Examples`. It defines the copy α *Set* of sets as a trivial quotient, transporting all relevant set operations to the new type. Theory `List_Cset.thy` sets up code generation via (standard) lists. The relevant code lemmas arise from the same equations on the set level. The code generation setup for distinct list or trees is not finised yet.

A user who wishes to execute sets as distinct lists implements his algorithm in terms of α *set*, making sure that only executable operations are used (in particular, unbounded set comprehension is not executable). Then, after importing `Dlist_Cset.thy`, the constants and their code equations can be transported to type α *Set* with the help of the quotient package. Since that type is executable by the library setup, no further configuration is needed for generating code.

From the code generation scenarios outlined in §1.1, the above applies to the program verification scenario. The other three scenarios work out of the box in case the term/proposition in question does not mention sets itself. This is the case when the sets are merely used as an intermediate data structure and do not show up in the interface of the algorithm itself. If sets are part of the interface, then the diagnostic commands *value* and *quickcheck*, as well as the evaluation oracle require some adaptation. Essentially, the term/proposition must be transported over to α *Set*, and the result (terms for *value* and *quickcheck* and a theorem for *evaluation*) must be transported back.

6 Work Items and Open Questions

This report concludes with a few things that could be done to improve the situation for data refinement and transport principles in general.

Streamlining data refinement with better tool setup

1. Using the quotient view of α *Set*, conduct a major case study, to see if the method really works for non-trivial cases. In particular, which knowledge of the concepts underlying quotients are required from a user, and which can be hidden?

2. Address the restrictions of the quotient package mentioned in §4.2. In particular, one wants to be able to configure the lifting machinery to work on quotient types that were defined by other means.

3. Implement a thin wrapper around the quotient package that allows presents a simpler interface to the data refinement setting. Document its usage.

4. A completely different approach could be to try to remove the restriction in the code generator that data refinement can only be applied to type constructors. The idea is that one promotes a type synonym to an actually type constructor in the code generator's intermediate language. Certain constants must then be annotated with special type signatures and will be translated to the new type. That new type takes the role of α *Set*, but it only exists in the intermediate language, not in the logic. In the rewriting semantics, this does not pose any problems, but technical issues arise from type classes: basically, we cannot have any class instances for the new type, since it does not exist from the logical point of view. In particular, this prevents the reification process (translating ML terms back into the logic) from working, which is based on a special type class.

   One could investigate if these issues could be solved somehow to support data refinement for open types as well. This would make the type α *Set* and the whole transport business obsolete, and users could use α *set* directly.

   Theory Reference. Based on a preliminary implementation of the above approach, theory `HOL/Library/Executable_Set.thy` makes sets executable in terms of lists. However, there is a big warning, telling users not to use this, duo to the problems sketched above.

Unifying the various transport tools in the system We have already seen that there is quite some overlap between *transfer* and the quotient package. Since neither tool completely subsumes the other, there is the obvious question whether there exists a common "supremum" which could replace both tools.

Starting from the quotient package, we are thus looking for the following generalizations:

Non-surjective morphisms The embedding from *nat* to *int* is not surjective, and thus we cannot reinterpret that embedding as a quotient. In principle, it should be possible to extend the tool to support quotients into a subset of some type, represented via an explicit predicate, but this is a nontrivial complication of the implementation. In particular, the equation $Abs\ (Rep\ x) = x$ no longer holds unconditionally. It is unclear whether this extension is practical.

Unlifting Currently, only one transport direction is supported by the quotient package. Facts are moved from the raw type to the quotient type. When we are interested in isomorphisms, the reverse direction is equally useful. It turns out that the package already proves the equivalence of both versions internally (except in the regularization step, which we can probably disregard). Thus it seems straightforward to export this functionality to the user in an "unlifting" transformation.

However, the story does not end here, as the Isabelle distribution contains at least two more implementations of transport functionality, which waits to be revisited and possibly subsumed by a general mechanism:

- To ease the transition between sets and bredicates, Berghofer implemented two attributes `[to_set]` and `[to_pred]`, which basically implement the (un)currying transformation needed to convert between relations of type $(\alpha_1 \times \cdots \times \alpha_n)$ *set* and predicates of type $\alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow$ *bool*. This is of course an isomorphism (or rather: a family of isomorphisms, one for each arity), and one would expect that there is quite a bit of overlap with the other tools.

- The formalization of nonstardard analysis contains another transfer tool that converts goals from nonstandard extensions of arbitrary types to the original type. This tool was developed by Huffman [3] and predates both the *transfer* tool discussed above (with which it shares the name, but no implementation) and the quotient package. The type of nonstardard extensions is actually defined (manually) as a quotient, and upon casual examination, it seems that the transfer principle is nothing but a specialized lifting procedure for that quotient. However, more investigation is required to find out if this is really the case. If it is, the development can probably be simplified considerably.

Theory Reference. The attributes `[to_set]` and `[to_pred]` are defined in `HOL/Tools/inductive_set.ML`. The nonstardard analysis transfer principle is set up in `HOL/NSA/StarDef.thy`. Huffman's paper [3] contains a detailed description.

Acknowledgements. Most of the knowledge and ideas presented here were collected in discussions with Amine Chaieb, Florian Haftmann, Cezary Kaliszyk, Peter Lammich, Tobias Nipkow, and Andrei Popescu. Cezary deserves special thanks for explaining many details about the quotient package.

References

[1] F. Haftmann. *Code Generation from Specifications in Higher-Order Logic.* PhD thesis, Institut für Informatik, Technische Universität München, Germany, 2009.

[2] F. Haftmann. Data refinement (Raffinement) in Isabelle/HOL, 2010. Draft manuscript.

[3] B. Huffman. Transfer principle proof tactic for nonstandard analysis. In *NETCA 2005*, 2005.

[4] C. Kaliszyk and C. Urban. Quotients revisited for Isabelle/HOL. In W. C. Chu, W. E. Wong, M. J. Palakal, and C.-C. Hung, editors, *ACM Symposium on Applied Computing (SAC'11)*, pages 1639–1644. ACM, 2011.

[5] A. Lochbihler. Formalising FinFuns - generating code for functions as data from Isabelle/HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 310–326. Springer Verlag, 2009.