Theorem Reuse by Proof Term Transformation

Einar Broch Johnsen[1] and Christoph Lüth[2]

[1] Department of Informatics, University of Oslo, Norway
einarj@ifi.uio.no
[2] FB 3 – Mathematics and Computer Science, Universität Bremen, Germany
cxl@informatik.uni-bremen.de

Abstract. Proof reuse addresses the issue of how proofs of theorems in a specific setting can be used to prove other theorems in different settings. This paper proposes an approach where theorems are generalised by *abstracting* their proofs from the original setting. The approach is based on a representation of proofs as logical framework proof terms, using the theorem prover Isabelle. The logical framework allows type-specific inference rules to be handled uniformly in the abstraction process and the prover's automated proof tactics may be used freely. This way, established results become more generally applicable; for example, theorems about a data type can be reapplied to other types. The paper also considers how to reapply such abstracted theorems, and suggests an approach based on mappings between operations and types, and on systematically exploiting the dependencies between theorems.

1 Introduction

Formal proof and development requires considerable effort, which can be reduced through reuse of established results. Often, a new datatype or theory resembles a previously developed one and there is considerable gain if theorems can carry over from one type to another. Previous work in this area addresses reuse by proof or tactic modification in response to changes in the proof goal such as modifying the constructors of a datatype, or unfortunate variable instantiations during a proof search [6, 7, 14, 22, 26]. In contrast, type-theoretic approaches [12, 13, 20] investigate the generalisation and modification of proofs by transforming the associated proof terms in the context of constructive type theory. This paper proposes a method for abstracting previously established theorems by proof transformations in a logical framework with proof terms. Logical frameworks are particularly well-suited for this approach, because inference rules are represented as formulae in the formalism; the choice of object logic becomes independent of the meta-logic in which the proof terms live.

The method we propose has been implemented in Isabelle [16], using the proof terms recently added by Berghofer and Nipkow [4]. Isabelle offers a wide range of powerful tactics and libraries, and we can work in any of the logics encoded into Isabelle, such as classical higher-order logic (HOL), Zermelo-Fraenkel set theory (ZF), and various modal logics [17]. However, the approach should be applicable to any logical framework style theorem prover.

K. Slind et al. (Eds.): TPHOLs 2004, LNCS 3223, pp. 152–167, 2004.
© Springer-Verlag Berlin Heidelberg 2004

The paper is organised as follows. Sect. 2 presents the different proof transformations of the abstraction method and Sect. 3 discusses how the transformations are implemented as functions on Isabelle's proof terms. Sect. 4 considers how to reuse abstracted theorems in a different setting, and demonstrates our approach in practice. Sect. 5 considers related work and we conclude in Sect. 6.

2 Generalising Theorems by Proof Transformation

This section proposes a method for abstracting theorems in logical frameworks by means of proof transformations, in order to derive generally applicable inference rules from specific theorems. A logical framework [8,19] is a meta-level inference system which can be used to specify other, object-level, deductive systems. Well-known examples of implementations of logical frameworks are Elf [18], λProlog [15], and Isabelle [17]. The work presented uses Isabelle, the meta-logic of which is intuitionistic higher-order logic extended with Hindley-Milner polymorphism and type classes.

In the logical framework, the formulae of an object logic is represented by higher-order abstract syntax and object logic derivability by a predicate on the terms of the meta-logic: Meta-level implication $\Longrightarrow$ reflects object level derivability. Object logics are represented by axioms encoding the axioms and inference rules of the object logic. The meta-logic is typed, with a special type *prop* of logical formulae (propositions); object logics extend the type system. The meta-level quantifier $\bigwedge$ can range over terms of any type, including *prop*. The logical framework allows us to prove theorems directly in the meta-logic. The correctness of all instantiations of a meta-logic theorem, or *schema*, follows from the correctness of the representation of the rules of the object logic. Theorems established in the meta-logic are derived inference rules of the object logic. Hence, new object logic inference rules can be derived within the logical language.

For the presentation of the abstraction method, we consider a proof π of a theorem ϕ, consisting of a series of inference steps in the meta-logic. The proposed generalisation process will transform π in a stepwise manner into a proof of a schematic theorem which may be instantiated in any other setting, i.e. a derived inference rule of the logic. The process consists of three phases:

1. making assumptions explicit;
2. abstracting function symbols;
3. abstracting type constants.

Each step in this process results in a proof of a theorem, obtained by transforming the proof of the theorem from the previous step. In order to replace function symbols by variables, all relevant information about these symbols, such as defining axioms, must be made explicit. In order to replace a type constant by a type variable, function symbols of this type must have been replaced by variables. Hence, each phase of the transformation assumes that the necessary steps of the previous phases have already occurred. The final step results in a proof π' from which we derive a schematic theorem $\psi \Longrightarrow \phi'$, where ϕ' is a modification of

the initial formula ϕ. In such theorems, the formulae of ψ are called *applicability conditions* as they identify theorems that are needed to successfully apply the derived rule. A necessary precondition for the second abstraction step is that the logical framework allows for higher-order variables, and for the third step that the logical framework allows for type variables.

It is in principle possible to abstract over all theorems, function symbols, and types occurring in a proof. However, such theorems are hard to use; for applicability, it is essential to strike a balance between abstracting too much and too little. Some tactics guiding the application of abstracted theorems are considered in Sect. 4.

2.1 Making Proof Assumptions Explicit

In tactical theorem provers such as Isabelle, the use of auxiliary theorems in a proof may be hidden to the user, due to the automated proof techniques. These contextual dependencies of a theorem can be made explicit by inspecting its proof term. In a natural deduction proof, auxiliary theorems can be introduced as leaf nodes in open branches of the proof tree.

Given an open branch with a leaf node theorem in the proof, we can close the branch by the implication introduction rule, thus transforming the conclusion of the proof. By closing all open branches in this manner, every auxiliary theorem used in the proof becomes visible in the root formula of the proof. To illustrate this process, let us reconsider the proof π of theorem ϕ. At the leaf node of an open branch π_i in the proof we find a theorem, say $\psi_i(x^i_1, \ldots, x^i_{k_i})$. We close the branch π_i by applying $\Longrightarrow$-introduction at the root of the proof, which leads to a proof of a formula $\forall x^i_1, \ldots, x^i_{k_i}\ \psi_i(x^i_1, \ldots, x^i_{k_i}) \Longrightarrow \phi$, where ψ_i has been transformed into a closed formula ψ'_i by quantifying over free variables, to respect variable scoping. The transformation of a branch is illustrated in Figure 1. This process is repeated for every branch in π with a relevant theorem in its leaf node. If we need to make j theorems explicit, we thereby derive a proof π' of the formula $(\psi'_1 \wedge \ldots \wedge \psi'_j) \Longrightarrow \phi$.

Generally, we may assume that a leaf node theorem is stronger than necessary for the specific proof. Therefore, it is possible to modify the applicability conditions of the derived theorem in order to make these easier to prove in a new setting. For example, if ψ_i is simplified by an elimination rule in the branch, we may (repeatedly) cut off the branch above the weaker theorem before closing the branch. Proofs in higher-order natural deduction can be converted into a normal form where all elimination rules appear above the introduction rules in each branch of the proof [21]. With this procedure, proofs on normal form result in the weakest possible applicability conditions, but proofs on normal form are not required for the abstraction process and proof normalisation is therefore not considered in this paper. Furthermore, if ψ_i is the leaf node theorem of an open branch in the proof π and all leaf node theorems in open branches in the proof of ψ_i are included among the leaf node theorems of other open branches of π, expanding π with the proof of ψ_i at appropriate leaf nodes before the proof transformation will remove superfluous applicability conditions from the derived

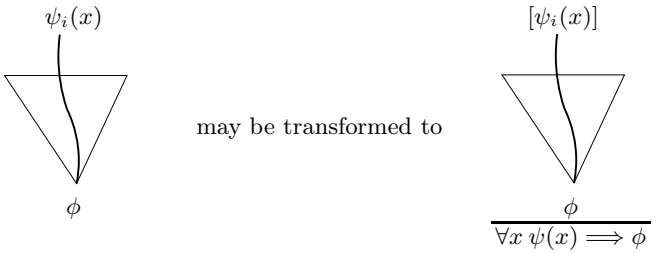

Fig. 1. The transformation and closure of a branch in the proof, binding the free variable x of the leaf node formula.

theorem. The usefulness of these improvements depends on the form of the proof and may cause considerable growth in the size of the proof term. An alternative is to consider the dependency graph between theorems (see Sect. 4.4). Our present approach is to transform the proof as it is given.

2.2 Abstracting Function Symbols

The next phase of the transformation process consists of replacing function symbols by variables. When all implicit assumptions concerning a function symbol F have been made explicit, as in the transformed theorem above, all relevant information about this function symbol is contained within the new theorem. The function symbol has become an *eigenvariable* because the proof of the theorem is independent of the context with regard to this function symbol. Such function symbols can be replaced by variables throughout the proof. Let $\phi[x/t]$ and $\pi[x/t]$ denote substitution, replacing t by x in a formula ϕ or proof π, renaming bound variables as needed to avoid variable capture.

A central idea in logical framework encodings is to represent object logic variables by meta-logic variables [19], which are placeholders for meta-logic terms. Hereafter, all free variables will be meta-variables and the abstraction process replaces function symbols by meta-variables. If the function symbol F is of type τ and a is a meta-variable of this type, the theorem $(\psi'_1 \wedge \ldots \wedge \psi'_i) \Longrightarrow \phi$ may be further transformed into

$$(\psi'_1[a/F] \wedge \ldots \wedge \psi'_i[a/F]) \Longrightarrow \phi[a/F], \tag{1}$$

by transforming the proof π' into a new proof $\pi'[a/F]$.

2.3 Abstracting Types

When all function symbols depending on a given type have been replaced by term variables, the name of the type is arbitrary. In fact, we can now replace such type constants by free type variables. The higher-order resolution mechanism of the theorem prover will then instantiate type variables as well as term variables when we attempt to apply the derived inference rule to a proof goal. However,

the formal languages used by theorem provers have structured types which may give rise to type-specific inference rules. When these occur in the proofs, they must also be made explicit for type abstraction to work. This is illustrated by the following example.

2.4 Example

We assume as object logic a higher-order equational logic, with axioms including symmetry (*sym*), reflexivity (*refl*), etc. In this object logic, consider a theory including the standard operations 0, S, $+$, and axioms defining addition and induction on the type N of natural numbers:

$$ax1 \equiv x + 0 = x \qquad ax2 \equiv x + Sy = S(x + y)$$
$$ind \equiv [\![p(0); \bigwedge t.p(t) \Longrightarrow p(St)]\!] \Longrightarrow p(x).$$

A proof of $x + 0 = 0 + x$ in this theory is as follows, slightly edited for brevity:

$$\dfrac{\dfrac{}{0+0=0+0}\,\textit{refl} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{St+0=S(t+0)}\,\textit{ax1} \qquad \dfrac{}{S(t+0)=0+St}\,\textit{ax2}}{St+0=0+St}\,\textit{trans}}{t+0=0+t \Longrightarrow St+0=0+St}\,\Longrightarrow\textit{-intro}_1}{\bigwedge t \cdot t+0=0+t \Longrightarrow St+0=0+St}\,\bigwedge\textit{-intro} \qquad \dfrac{}{\vdots}\,\textit{ind}}{x+0=0+x}\,\Longrightarrow\textit{-elim} \tag{2}$$

Applying the first step of the abstraction method, all theorems from the theory become assumptions, which results in a proof of the following theorem:

$$[\![[\![p(0); \bigwedge t.p(t) \Longrightarrow p(St)]\!] \Longrightarrow p(x); x + 0 = x; x + Sy = S(x + y))]\!] \Longrightarrow x + 0 = 0 + x$$

In the second and third step of the process, we first replace 0, S, and $+$ by the meta-variables a, b, and c, respectively. When this is done, we can replace the type constant N with a free type variable α, resulting in a proof of the theorem

$$[\![[\![p(a); \bigwedge t.p(t) \Longrightarrow p(b(t))]\!] \Longrightarrow p(x); c(x,a) = x; c(x,b(y)) = b(c(x,y))]\!] \Longrightarrow c(x,a) = c(a,x),$$

which can be applied as an inference rule to a formula of any type. In order to discharge the applicability conditions of the inference rule, the formula representing the induction rule must be a theorem for the new type.

3 Implementation of the Abstraction Techniques

Under the Curry-Howard isomorphism, proofs correspond to terms in a typed λ-calculus. We have implemented the abstraction processes from Sect. 2 in the

theorem prover Isabelle, which records proofs as meta-logic proof terms. The user can use all of Isabelle's automatic and semi-automatic proof infrastructure and Isabelle automatically constructs the corresponding meta-logic proof term [4]. Given a proof term, a theorem may be derived by replaying the meta-logic inference rules. We use this facility to derive new theorems: Given a theorem to abstract, we obtain its proof term, perform appropriate transformations on the proof term and replay the derived proof term to obtain a generalised theorem. Hence, the correctness of the derived theorem is guaranteed by the Isabelle's replay facility for proof terms. The implementation of abstraction functions does not impose any restrictions on the proof or the theorem: The abstraction process can be applied to any theorem, including those from Isabelle's standard libraries.

3.1 Proof Terms

This section introduces Isabelle's proof terms, which may be presented as

$$p ::= h \mid c_{[\tau_n/\alpha_n]} \mid \boldsymbol{\lambda} h : \phi.p \mid \lambda x :: \tau.p \mid p \cdot p \mid p\, t \tag{3}$$

where h, c, x, t, ϕ, α, and τ denote proof variables, proof constants, term variables, terms of arbitrary type, propositions, type variables, and types, respectively. The language defined by (3) allows for abstraction over term and proof variables, and application of proofs and terms to proofs, corresponding to the introduction and elimination of $\bigwedge$ and $\Longrightarrow$. Proof terms live in an environment which maps proof variables to terms representing propositions and term variables to their type. Proof constants correspond to axioms or already proved theorems. For more details and formal definitions, including the definition of provability in this setting, see [4].

Proof terms can be illustrated by the example of Proof (2). We identify theorem names with proof constants: *ax1*, *ax2*, *refl*, etc. The leftmost branch of the proof consists of the axiom $\mathit{refl} \equiv x = x$, with x instantiated by 0. This is reflected by the proof term

$$\pi_1 = \mathit{refl}\ 0.$$

The middle branch introduces a meta-implication in the proof term

$$\pi_2 = (\boldsymbol{\lambda} H : (\bigwedge x : N.x + 0 = 0 + x).\ \psi),$$

where ψ represents the body of the proof term (omitted here). The proof variable H represents an arbitrary proof of the proposition $\bigwedge x : N.x + 0 = 0 + x$ and is introduced by proof term $\boldsymbol{\lambda}$-abstraction. We can refer to a proof of this proposition in the proof term ψ by the proof variable H. The whole proof term for (2) becomes

$$\pi = \mathit{ind}\,(\lambda u.\ u + 0 = 0 + u)\ x \cdot \pi_1 \cdot \pi_2.$$

The premises π_1 and π_2, which correspond to the base case and the induction step, are applied to the induction rule *ind*, reflecting elimination of meta-implication. In contrast to proof level $\boldsymbol{\lambda}$-abstraction, term level λ-abstraction allows the higher-order variable p in *ind* to be instantiated with $\lambda u.\ u+0 = 0+u$.

3.2 Implementing Abstraction by Proof Term Transformations

The abstractions presented in Sect. 2 are implemented as functions which take theorems to theorems by transforming proof terms.

In proof terms, assumptions are represented by proof constants corresponding to previously proved theorems. For example, the proof term π above contains proof constants *ax1*, *ax2* and *ind*, which can now be lifted to applicability conditions as described in Sect. 2.1. This is done by adding a proof term $\boldsymbol{\lambda}$-abstraction outside the proof term and replacing occurrences of the theorem inside the proof term with an appropriate variable. After abstraction over *ax1* and *ax2* (omitting the lengthy but similar abstraction over *ind*), we obtain the proof term

$$\phi = \boldsymbol{\lambda} H : (\bigwedge x\, y : N. x + Sy = S(x + y)). \boldsymbol{\lambda} H' : (\bigwedge x : N. x + 0 = x). \pi[ax1/H', ax2/H]$$

which can be replayed to yield the following theorem:

$$[\![\forall x, y \cdot x + Sy = S(x + y); \forall x \cdot x + 0 = x]\!] \Longrightarrow x + 0 = 0 + x$$

Internally, deBruijn indices are used for bound variables, which explains the occurrence of H' in the second proof term. This gives a first simple version of the theorem abstraction function: traverse the proof tree, replace all nodes referring to the theorem we want to abstract over with the appropriate deBruijn index, and add a $\boldsymbol{\lambda}$-abstraction in front of the proof term.

When we use a theorem in a proof, both schematic and type variables are instantiated. If we make the theorem an applicability condition we need to quantify over both the schematic and type variables, hence the meta-quantification in H and H' above. However, abstraction over type variables is not possible in the Hindley-Milner type system of Isabelle's meta-logic, where type variables are always implicitly quantified at the outermost level. Instead, distinct assumptions must be provided for each type instance. For example, a proof of the theorem

$$map\,(f \cdot g)\,x = map\,f\,(map\,g\,x), \tag{4}$$

contains three different type instances of the definition of map for non-empty lists $map\,f\,(Cons\,x\,y) = Cons\,(f\,x)\,(map\,f\,y)$.

At the implementation level, abstracting operations (Sect. 2.2) and types (Sect. 2.3) is more straightforward. Traversing the proof term we replace operations and types by schematic and type variables, respectively. When abstracting over polymorphic operations, we need distinct variables for each type instance of the operation symbol, similar to the theorems above. If we consider `map` in Theorem (4), we need to abstract over each of the three type instances separately, resulting in three different function variables.

3.3 Abstraction over Theories

The previously defined *elementary abstraction functions* operate on single theorems, operations, and types. For a more high-level approach, *abstraction tactics* may be defined, which combine series of elementary abstraction steps.

An example of such a tactic is *abstraction over theories*. A theory in Isabelle can be thought of as a signature defining type constructors and operations, and a collection of theorems. Theories are organised hierarchically, so all theorems established in ancestor theories remain valid.

The tactic abstracts a theorem which belongs to a theory T_1 into an ancestor theory T_2. It collects all theorems, operations, and types from the proof term which do not occur in T_2, and applies elementary tactics recursively to abstract over each, starting with theorems and continuing with function symbols and types. Finally, the derived proof term is replayed in the ancestor theory, thus establishing the validity of the abstracted theorem in the theory T_2 [1].

Abstraction over all theorems, function symbols, and types will generally lead to theorems which are hard to reuse. In the next section, we will consider tactics which aid in the abstraction and reuse of abstracted theorems.

4 Reapplying Abstracted Theorems

This section considers different examples of abstraction, and scenarios to reapply abstracted theorems. As part of our experimentation with the abstraction method, we have generalised approximately 200 theorems from Isabelle's libraries, and reapplied these. A systematic approach to reapplication is suggested in order to facilitate reuse of sets of theorems.

4.1 Simple Abstraction and Reuse

A simple example of abstraction and reuse is to derive a theorem about natural numbers by abstraction from the theorem `append_Nil2` $\equiv x @ [\,] = x$ about lists. Applying the abstraction tactic `abs_to_thy` described in Sect. 3.3, we derive a theorem independent of the theory of lists:

$$\begin{array}{l} [\![\forall P\ l.\ [\![P\ nil\ ;\ \forall a\ l.\ P\ l \Longrightarrow P\ (cons\ a\ l)]\!] \Longrightarrow P\ l; \\ \quad \forall y.\ app\ nil\ y = y\ ; \\ \quad \forall u\ x\ y.\ app\ (cons\ u\ x)\ y = cons\ u\ (app\ x\ y)]\!] \\ \qquad\qquad \Longrightarrow app\ x\ nil = x \end{array} \tag{5}$$

The abstraction process introduces new names for the constant $[\,]$ and the infix operator @, favouring lexically suitable variable names.

We can now use this theorem to show that $x + 0 = x$. We proceed in two stages: we first instantiate the variables *nil* with 0, *app* with + and *cons* with $\lambda x.Suc$ (note we need the vacuous argument x here). This yields

$$\begin{array}{l} [\![\forall P\ l.\ [\![P\ 0;\ \forall a\ l.\ P\ l \Longrightarrow P\ (Suc\ l)]\!] \Longrightarrow P\ l; \\ \quad \forall y.\ y + 0 = y; \\ \quad \forall u\ x\ y.\ Suc\ x + y = Suc(x + y)\ ]\!] \Longrightarrow x + 0 = x. \end{array}$$

[1] Due to Isabelle's typeclasses an operation which is defined in T may not occur in the signature of T directly; in this case, the user has to explicitly give the operation.

The premises correspond to well-known theorems about natural numbers (induction and the definition of +). Resolving with these, we obtain the theorem $x + 0 = x$. Apart from the small simplification step required by moving from a parametric to a non-parametric type, this process can be completely automated.

4.2 Change of Data Representation

A more challenging situation occurs when we want to implement a datatype by another one. For example, suppose we implement the unary representation of natural numbers by a binary representation, which may be given as follows:

$$\begin{array}{ll} \textit{datatype } \textit{bNat} = & \textit{datatype } \textit{Pos} = \\ \quad \textit{Zero} & \quad \textit{One} \\ \quad \mid \textit{PBin Pos} & \quad \mid \textit{Bit Pos bool} \end{array}$$

The standard functions on *Nat* may be defined by means of bit operations in the binary number representation. We first define the successor functions *bSucc* on *bNat* and *pSucc* on *Pos* by primitive recursion. The latter is defined as

$$\begin{array}{l} \textit{pSucc One} = \textit{Bit One False} \\ \textit{pSucc (Bit x b)} = \textit{if b then Bit (pSucc x) False} \\ \qquad\qquad\qquad\qquad\quad \textit{else Bit x True} \end{array}$$

Subsequently, we define binary addition *bPlus* by primitive recursion by

$$\begin{array}{l} \textit{bPlus Zero x} = \textit{x} \\ \textit{bPlus (Pbin x) y} = \textit{(case y of Zero} \Rightarrow \textit{Pbin x} \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \textit{Pbin y} \Rightarrow \textit{Pbin (pPlus x y))} \end{array}$$

For *Pos*, we get:

$$\begin{array}{l} \textit{pPlus One y} = \textit{pSucc y} \\ \textit{pPlus (Bit x b1) y} = \\ \qquad \textit{(case y of One} \Rightarrow \textit{ pSucc (Bit x b1)} \\ \qquad\qquad \mid \textit{(Bit z b2)} \Rightarrow \textit{Bit (pPlus x (if (b1 \& b2) then pSucc z} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{else z))} \\ \qquad\qquad\qquad\qquad\qquad\quad (b1 \neq b2) \end{array}$$

We show how to prove $bPlus\ x\ Zero = x$ by reusing the abstracted form (5) of theorem `append_Nil2` ($xs\ @\ Nil = xs$). We instantiate, this time *nil* with *Zero*, *app* with *bPlus* and *cons* with $\lambda x.bSucc$, and obtain the theorem

$$\begin{array}{l} [\![\forall P\ l. [\![P\ Zero; \forall a\ l.\ P\ l \Longrightarrow P\ (bSucc\ l)]\!] \Longrightarrow P\ l; \\ \quad \forall x.\ bPlus\ Zero\ x = x; \\ \quad \forall u\ x\ y.\ (bSucc\ x)\ y = bSucc\ (bPlus\ x\ y)]\!] \Longrightarrow\ bPlus\ x\ Zero = x \end{array} \tag{6}$$

The first premise corresponds to the induction scheme, and the second and third premises correspond to the primitive recursive definition of addition on natural numbers. The induction principle on *Pos* is given by the structure of the

datatype, not by natural induction. For the first premise, we therefore need to show that the usual natural induction rule can be derived for *bNat*. This is done by first establishing an isomorphism between `Nat` and `bNat`, i.e. two functions $n2b : nat \longrightarrow bNat$ and $b2n : bNat \longrightarrow nat$ which are shown to be mutually inverse. The second premise is given by the definition of *bPlus*. The third premise can be proved through case analysis on x and y.

We next show how to prove *bPlus x (bSucc y) = bSucc (bPlus x y)* by reusing the proof of $\forall m\ n.\ m + Suc\ n = Suc\ (m + n)$ from the theory of natural numbers. The abstraction tactic gives us the theorem:

$$\begin{array}{l}[\![\ \forall P\ n.\ [\![P\ zero;\ \forall n.\ P\ n \Longrightarrow P\ (suc\ n)]\!] \Longrightarrow P\ n;\\ \quad \forall n.\ plus\ zero\ n = n;\\ \quad \forall u\ n.\ plus\ (suc\ u)\ n = suc\ (plus\ u\ n)\ ]\!]\\ \qquad\qquad \Longrightarrow plus\ m\ (suc\ n) = suc\ (plus\ m\ n)\end{array}$$

Instantiation (*zero* with *Zero*, *plus* with *bPlus*, *suc* with *bSucc*) yields a theorem with three premises, which are identical to the premises of (5), except that there are no vacuous quantified variables in the first and third premise. Hence, resolution with the theorems needed above directly proves the goal.

4.3 Moving Theorems Along Signature Morphisms

In the previous examples, the process of moving theorems from the theory `Nat` to `bNat` is quite mechanical: take a theorem from `Nat`, abstract all operations and types from `Nat`, then instantiate the resulting variables with the corresponding operations from `bNat`. In general, we can move theorems from a source theory to a target theory if there is a suitable mapping of types and operations between the theories. Such mappings between types and operations are known as *signature morphisms*.

A *signature* $\Sigma = \langle T, \Omega \rangle$ is given by *type constructors* T, with *arity* $ar_T : T \to \mathbb{N}$, and *operations* Ω, with arity $ar_\Omega : \Omega \to T^*$. T^* is the set of all well-formed types built from the type constructors and a (finitely countable) set of type variables. A signature morphism is a map between type constructors and operations preserving the arities of the type constructors, and the domain and range of the operations. Formally, given two signatures $\Sigma_1 = \langle T_1, \Omega_1 \rangle$ and $\Sigma_2 = \langle T_2, \Omega_2 \rangle$, a *signature morphism* $\sigma : \Sigma_1 \to \Sigma_2$ is given by a map $\sigma_T : T_1 \to T_2$ on type constructors and a map $\sigma_\Omega : \Omega_1 \to \Omega_2$ on operation symbols, such that

$$\forall \tau \in T_1.\ ar_{T_1}(\tau) = ar_{T_2}(\sigma_T(\tau)) \tag{7}$$

$$\forall \omega \in \Omega_1.\ \overline{\sigma}_T(\omega) = \sigma_\Omega(\omega) \tag{8}$$

where $\overline{\sigma}_T : T_1^* \to T_2^*$ is the unique extension of σ_T to all well-formed types. A *partial* signature morphism is given by partial maps $\sigma_T : T_1 \rightharpoonup T_2$ and $\sigma_\Omega : \Omega_1 \rightharpoonup \Omega_2$ such that all type constructors appearing in the source of any operation in the domain of σ_Ω are in the domain of σ_T.

Let $\mathtt{Thy}_1$ and $\mathtt{Thy}_2$ be Isabelle theories with signatures $\Sigma(\mathtt{Thy}_1)$ and $\Sigma(\mathtt{Thy}_2)$, and let $\sigma : \Sigma(\mathtt{Thy}_1) \to \Sigma(\mathtt{Thy}_2)$ be a signature morphism. Any proof term from

$\texttt{Thy}_1$ can be translated into a proof term in $\texttt{Thy}_2$ if the proof does not contain references to theorems from $\texttt{Thy}_1$. This gives us a canonical way of moving theorems from $\texttt{Thy}_1$ to $\texttt{Thy}_2$: first abstract all theorems from $\texttt{Thy}_1$ occuring in the proof of the theorem, then replace type constructors τ with $\sigma_T(\tau)$, and all operation symbols ω with $\sigma_\Omega(\omega)$, and replay the proof. Conditions (7) and (8) ensure that the translated proof term is well-typed.

In order to extend the implementation of the theorem reuse method with mappings of this kind, we define an abstract ML type `sig_morph` for partial signature morphisms. Signature morphisms are obtained using a constructor which checks the conditions (7) and (8) above, when given source and target signatures and the function graphs. We can apply the signature morphism in order to map types and terms from the source into the target signature. The invariants of the signature morphisms make sure that a translated term typechecks if the original term did. Given this type, we can define an abstraction tactic

```
val abs_translate : sig_morph-> thm-> thm
```

which moves a theorem along a signature morphism. Applying this abstraction tactic to our example, we can move any theorem from `Nat` to `bNat`, such as the theorems `add_0_right`, `add_Suc_right` (see Sect 4.2), and `add_commute`:

$$\begin{array}{l}[\![\ \forall P\ n.\ [\![\ P\ Zero;\ \forall n.\ P\ n \Longrightarrow P\ (bSucc\ n)\ ]\!] \Longrightarrow P\ n;\\ \quad \forall n.\ bPlus\ Zero\ n = n;\ \forall m.\ bPlus\ m\ Zero = m;\\ \quad \forall u\ n.\ bPlus\ (bSucc\ u)\ n = bSucc\ (bPlus\ u\ n);\\ \quad \forall m\ n.\ bPlus\ m\ (bSucc\ n) = bSucc\ (bPlus\ m\ n)\ ]\!]\\ \qquad \Longrightarrow bPlus\ m\ n = bPlus\ n\ m\end{array}$$

In the translated theorems, the applicability conditions correspond to theorems that were used in the proof of the source theorems. This suggests that we can partially automate discharge of the applicability conditions when moving several theorems from a source theory to a target theory by considering the order in which the theorems are established in the source theory.

4.4 Analysing Theorem Dependencies

This section considers how to reduce proof work when moving theorems between theories. In the previous examples, we have seen that it was necessary to prove certain applicability conditions in the derived theorems. Some applicability conditions occur in several theorems (e.g. the induction rule above) and a derived theorem may occur as an applicability condition in another. Proof of applicability conditions may be considerably simplified by analysis of the source theory prior to theorem reuse.

In the example of `Nat` and `bNat`, successor and addition for `bNat` were defined in terms of bit operations. This resulted in applicability conditions to ensure that the definition of addition in `Nat` was valid in `bNat`. In general, we would like to identify an appropriate, small set of theorems that need manual proof in a target theory in order to move a larger selection of theorems from the source theory to

the target theory automatically. We shall call such a set of theorems an *axiomatic base*. Finding an axiomatic base is a process which is hard to automate; for the `Nat` example above, the axiomatic base is the Peano axioms and the definition of addition. We will below give an algorithm which checks if a given set of theorems form an axiomatic base, and if so provides an abstraction tactic to move across theorems automatically. Isabelle's visualisation tool for the dependency graph may help determine an appropriate axiomatic base.

We say that a theorem φ *depends on* another theorem ψ, written $\psi \longrightarrow \varphi$, if ψ occurs as a leaf node in the proof of φ. The *premises* $prem(\varphi)$ of a theorem φ is the set of all theorems on which the theorem depends, i.e. $prem(\varphi) = \{\psi \mid \psi \longrightarrow \varphi\}$. This allows the construction of a *dependency graph* for a theory, in which the nodes are theorems of the theory, and the (unlabelled) edges are $\psi \longrightarrow \phi$ for theorems ψ and ϕ. The dependency graph of the source theory helps to identify an appropriate axiomatic base.

Given a set Φ of theorems, let $pre(\Phi)$ denote the *preconditions* of Φ, i.e. the set of all theorems needed to derive the theorems in Φ. This set can be obtained from the dependency graph by a simple depth- or breadth-first search:

$$pre(\Phi) = \Phi \cup pre(\{\psi \mid \psi \longrightarrow \varphi \text{ for } \varphi \in \Phi\}).$$

A theorem φ is *directly derivable* from a set Ψ of theorems, written $\Psi \vdash \varphi$, if all its premises (in the theory) are contained in Ψ:

$$\Psi \vdash \varphi \Leftrightarrow prem(\varphi) \subseteq \Psi.$$

This means that if we have translated all theorems in Ψ, we can establish the translation of φ by replaying its translated proof term. The set of theorems *derivable* by proof replay from a set of theorems Φ is the closure under derivability:

$$der(\Phi) = \Phi \cup der(\{\psi \mid \Phi \vdash \psi\}).$$

Given a source theory with a set Φ of theorems and an axiomatic base B, a target theory, and a partial signature morphism between the theories, we can systematically abstract all theorems in the set

$$A \stackrel{def}{=} pre(\Phi) \setminus pre(B)$$

and instantiate according to the signature morphism, deriving theorems in the target theory. A necessary condition for the success of this translation is that $A \subseteq der(B)$, i.e. the axiomatic base is strong enough to derive the theorems in Φ. In order to move theorems to another theory, the theorems of the axiomatic base, translated according to the signature morphism, must be proved in the target theory. In the example of Sect. 4.2, if Φ includes `add_commute`, the axiomatic base will typically include the Peano axioms for addition, but need not include `add_0_right`, which is derivable from these axioms. If the theorems of Φ are translated in the order of dependency, such that the premises $prem(\varphi)$ are moved before φ for all $\varphi \in A$, the applicability conditions of the derived theorems in the target theory can be discharged automatically.

Our implementation provides a module which implements the necessary graph algorithms. For simplicity and speed, we refer to theorems by their name throughout. In particular, the function

```
val saturate : DG -> string list-> string list
                              -> (string* string list) list
```

will, given a dependency graph, an axiomatic base and a list of theorems, provide a list of pairs of theorems and the names of their premises in order of dependency. This list can be given to the abstraction tactic which moves across the theorems.

5 Related Work

The problem of proof reuse has been addressed previously. Some approaches apply branches or fragments from old proofs when solving new problems: Melis and Whittle [14] study reasoning by analogy, a technique for reusing problem solving experience by proof planning; Giunchiglia, Villefiorita, and Walsh [7] study abstraction techniques, where a problem is translated into a related abstract problem which should be easier to prove as irrelevant details are ignored; and Walther and Kolbe, in their PLAGIATOR system [26], suggest proof reuse in first-order equational theories by so-called proof catches, a subset of the leaf nodes in a proof tree, similar to our applicability conditions.

The KIV system reuses proof fragments to reprove old theorems after modifications to an initial program [22]. The approach exploits a correspondence between positions in a program text and in the proofs, so that subtrees of the original proof tree can be moved to new positions. This depends on the underlying proof rules, so the approach is targeted towards syntax-driven proof methods typical of program verification. A more semantic approach are development graphs as implemented in MAYA [3], where a specification is represented by a development graph, a richer version of the dependency graphs from Sect. 4.4.

In a logical framework setting, Felty and Howe [6] describe a generic approach to generalisation and reuse of tactic proofs. In their work, a proof is a nested series of proof steps which may have open branches. Reuse is achieved by replacing the substitutions of a proof with substitutions derived from a different proof goal by means of higher-order resolution. This opens for an elegant way to reuse steps from abortive proof attempts for e.g. unfortunate variable instantiations, which can to some extent be mimicked by considering different unifiers for our derived inference rules. In contrast to the cited works, our approach allows a generalisation over types as well as function symbols. In particular, proof reuse as in the examples of Sect. 4 is not feasible in the cited approaches.

Proof reuse and generalisation of theorems have been studied in the Coq system. Proofs in Coq resemble proof terms in Isabelle, but in a richer type theory. Pons, Magaud and Bertot [13,20] consider transformation of proofs, similar to ours, replacing operation symbols and types by variables, and Magaud and Bertot [13] consider change of data representation in this setting, studying in particular the same example as in Sect. 4.2 (in fact, our example was inspired by

this work), extended further to type-specific inference rules in [12]. The richer type theory used by Coq makes proof term manipulation more involved than in the logical framework setting of our approach. For reuse, proofs have to be massaged into a particular form, and e.g. induction and case distinction have to be given special treatment. There are particular methods which either generalise theorems [13, 20], or handle change of data type representations [12, 13]. In our approach induction and case distinction are represented as meta-logic axioms, which allows a uniform treatment of these situations by appropriate abstraction tactics. For example, the dependency analysis (Sect. 4.4) is built on top of the more elementary abstraction and reuse tactics. Further, theorems abstracted with our method may be instantiated several times in different settings, thus allowing multiple reuse.

Dependency graphs and similar structures have been considered in systems such as KIV or Maya [3]. Isabelle can visualise the dependency graph, but not in an interactive way. More interesting here is the work by Bertot, Pons and Pottier [5], who implemented an interactive visualisation of the dependency graph, allowing manipulations such as removing and grouping of edges and labels. An interactive tool in this vein would greatly aid the user in establishing an axiomatic base.

6 Conclusion and Future Work

This paper demonstrates how theorems can be generalised from a given setting and reapplied in another, exploiting the possibilities offered by proof terms in logical framework style theorem provers. This approach combines proof term manipulation as known from type theory with the flexibility and power of logical frameworks. The combination is particularly well suited for changing data representations because object logic inference rules and theorems may be given a uniform treatment in both the abstraction and reuse process. Consequently, the transformation method may be applied to any theorem in a direct way, allowing multiple reuse of the abstracted theorem in different settings.

The considered strategies for reuse point in interesting directions. Signature morphisms are used as a structuring mechanism in algebraic specification languages such as CASL [2], and for structured development in e.g. Maya [3] or Specware [23, 25]. The proposed analysis of theorem dependencies is promising, and should be supported by a (probably graphical) tool, which would allow the user to interactively determine an axiomatic base for the theory, assisted by appropriate heuristics.

In addition to theorem reuse as discussed in this paper, the proposed method may have applications in formal program development. In this field, several approaches have been suggested based on specialised transformation rules [23, 24] or deriving rules from theorems [1, 11]. However a coherent framework is lacking, allowing users to systematically generalise existing developments to a widely applicable set of transformation rules. The proposed abstraction method may be of use here. A small demonstration of this application, deriving transformation rules from correctness proofs of data refinements, may be found in [10].

The suggested proof term transformations and reuse strategies have been implemented in Isabelle 2003[2]. The implementation comprises only about one thousand lines of ML code, with the abstraction tactics accounting for roughly 40%, dependency analysis and signature morphisms about 30%, and auxiliaries and utilities the rest. The compactness of the code suggests that the framework of meta-logic proof terms provided by Isabelle is well-suited for this kind of transformations. At a technical level, there are several ways of improving the proposed abstraction method. An interesting improvement is to incorporate the technique of coloured terms [9], which would allow several variables to replace the same function symbol during abstraction.

Acknowledgements

Part of this work was done while the first author was affiliated with the University of Bremen. The work was supported by the DFG under grant LU 707-1. We are grateful to Till Mossakowski and Burkhart Wolff for interesting discussions on the subject of abstraction in Isabelle, to Erwin R. Catesbeiana for pointing out potential pitfalls, and to Stefan Berghofer for explaining some technicalities of Isabelle's proof terms to us.

References

1. P. Anderson and D. Basin. Program development schemata as derived rules. *Journal of Symbolic Computation*, 30(1):5–36, July 2000.
2. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.
3. S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The development graph manager MAYA. In H. Kirchner and C. Ringeissen, editors, *Proc. 9th Int. Conf. Algebraic Methodology and Software Technology (AMAST'02)*, LNCS 2422, p. 495–501. Springer, 2002.
4. S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *13th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'00)*, LNCS 1869, p. 38–52. Springer, 2000.
5. Y. Bertot, O. Pons, and L. Rideau. Notions of dependency in proof assistants. In *User Interfaces in Theorem Provers*, Eindhoven Univ. of Technology, 1998.
6. A. Felty and D. Howe. Generalization and reuse of tactic proofs. In F. Pfenning, editor, *5th Intl. Conf. on Logic Programming and Automated Reasoning (LPAR'94)*, LNCS 822, pages 1–15. Springer, July 1994.
7. F. Giunchiglia, A. Villafiorita, and T. Walsh. Theories of abstraction. *AI Communications*, 10(3-4):167–176, 1997.
8. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993.

[2] The source code and examples of this paper may be downloaded from `http://www.informatik.uni-bremen.de/~cxl/sources/tpholsO4.tar.gz`

9. D. Hutter and M. Kohlhase. Managing structural information by higher-order colored unification. *Journal of Automated Reasoning*, 25:123–164, 2000.
10. E. B. Johnsen and C. Lüth. Abstracting refinements for transformation. *Nordic Journal of Computing*, 10(4):313–336, 2003.
11. C. Lüth and B. Wolff. TAS – a generic window inference system. In J. Harrison and M. Aagaard, editors, *13th Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs'00)*, volume LNCS 1869, p. 405–422. Springer, 2000.
12. N. Magaud. Changing data representation within the Coq system. In *16th Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs'03)*, LNCS 2758, p. 87–102. Springer, 2003.
13. N. Magaud and Y. Bertot. Changing data structures in type theory: A study of natural numbers. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs, Intl. Workshop (TYPES 2000)*, LNCS 2277, p. 181–196. Springer, 2002.
14. E. Melis and J. Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22(2):117–147, 1999.
15. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
16. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, LNCS 2283. Springer, 2002.
17. L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, p. 361–386. Academic Press, 1990.
18. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, p. 149–181. Cambridge Univ. Press, 1991.
19. F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, p. 1063–1147. Elsevier Science Publishers, 2001.
20. O. Pons. Generalization in type theory based proof assistants. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs, Intl. Workshop (TYPES 2000)*, LNCS 2277, p. 217– 232. Springer, 2002.
21. D. Prawitz. Ideas and results in proof theory. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, *Studies in Logic and the Foundations of Mathematics* 63, p. 235–307. North-Holland, Amsterdam, 1971.
22. W. Reif and K. Stenzel. Reuse of proofs in software verification. In R. K. Shyamasundar, editor, *Proc. Foundations of Software Technology and Theoretical Computer Science*, LNCS 761, p. 284–293. Springer, 1993.
23. D. Smith. Constructing specification morphisms. *Journal of Symbolic Computation*, 15:571– 606, 1993.
24. D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14:305– 321, 1990.
25. Y. V. Srinivas and R. Jullig. Specware: Formal support for composing software. In *Proc. Conf. Mathematics of Program Construction*, LNCS 947. Springer, 1995.
26. C. Walther and T. Kolbe. Proving theorems by reuse. *Artificial Intelligence*, 116(1–2):17–66, 2000.