Nordic Journal of Computing 10(2003), 313–336.

ABSTRACTING REFINEMENTS FOR TRANSFORMATION

EINAR BROCH JOHNSEN
University of Oslo, Department of Informatics
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
einarj@ifi.uio.no

CHRISTOPH LÜTH
Universität Bremen, FB 3 — Mathematik und Informatik
P.O. Box 330 440, D-28334 Bremen, Germany
cxl@informatik.uni-bremen.de

Abstract. Formal program development by stepwise refinement involves a lot of work discharging proof obligations. Transformational techniques can reduce this work: applying correct transformation rules removes the need for verifying the correctness of each refinement step individually. However, a crucial problem is how to identify appropriate transformation rules.

In this paper, a method is proposed to incrementally construct a set of correctness preserving transformation rules for refinement relations in arbitrary specification formalisms. Transformational developments are considered as proofs, which are generalised. This results in a framework where specific example refinements can be systematically generalised to more applicable transformation rules. The method is implemented in the Isabelle theorem prover and demonstrated on an example of data refinement.

ACM CCS Categories and Subject Descriptors: F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic; I.2.2 [Artificial Intelligence]: Automatic Programming

Key words: program refinement, transformation, theorem provers, proof reuse

1. Introduction

When a program is developed from a specification by stepwise refinement, each refinement step has to be proved correct, incurring a lot of work discharging proof obligations. We believe that the development process can be effectively supplemented by transformational techniques. The work load is significantly reduced by applying transformation rules which have been proven correct previously. This way, developers only need to select appropriate rules in order to apply standard development steps, and can consequently focus on the non-standard steps of the development process containing the non-trivial design decisions. However, the advantage gained from transformational techniques depends on the ability to construct an appropriate set of transformation rules. In this paper, we propose a

Received March 17, 2003; accepted August 22, 2003.

method to incrementally construct a set of transformation rules for refinement relations in arbitrary specification formalisms by generalising existing, concrete refinement steps.

Good transformation rules should strike a balance between wide applicability and ease of verification. In particular, the proof obligations should be substantially easier to prove than the refinement directly. Libraries of tailored transformation rules have been developed for particular systems with built-in notions of refinement [Bauer *et al.* 1985, Hoffmann and Krieg-Brückner 1993, Smith 1990]. How can similar libraries of efficient and correctness preserving transformation rules be developed to capture refinement in arbitrary specification formalisms, for example Z [Woodcock and Davies 1996, Derrick and Boiten 2001] or CSP [Roscoe 1998]?

Our approach to this problem is based on the close correspondence between proof obligations for refinement and applicability conditions for transformation schemata: transformational development corresponds to proving theorems. This correspondence gives us an incremental method for constructing transformation rules: refinements may be formulated as theorems and generalised by means of proof transformations. The setting of this work is the generic transformation system TAS [Lüth and Wolff 2000], based on the Isabelle theorem prover [Nipkow *et al.* 2002]. In contrast to related systems, TAS can be instantiated with any specification formalism which is encoded into Isabelle and supports a notion of refinement. Using TAS, transformation rules become Isabelle tactics, i.e. programmed proof procedures. Isabelle is used both as a logical framework, guaranteeing correctness and consistency, and as a powerful proof engine to discharge proof obligations by e.g. providing automatic induction and simplification techniques for user-defined datatypes.

With a generic system, established and popular specification formalisms can be combined with a transformational development methodology. This results in a tool for stepwise refinement of specifications in such formalisms as Z or CSP. However, a method is needed to develop (libraries of) transformation rules for the refinement relations of such specification languages. For this purpose, we suggest a method to generalise theorems by abstracting them from the assumptions (i.e., other theorems), operations, and types used in the proof of the theorem. With this method, new transformation rules may be derived in a systematic manner by generalising the theorems corresponding to concrete refinement steps.

The approach is illustrated by the technique of data refinement [Hoare 1972, de Roever and Engelhardt 1998, Derrick and Boiten 2001], which is used in many specification languages. We formalise data refinement in Isabelle and a specific refinement proof is generalised to provide a transformation rule. The approach can be applied to other formal notions of refinement and other specification formalisms, given an Isabelle encoding. We believe that, with appropriate sets of transformation rules, systems such as TAS can become useful assistants for formal program development in already established and widely used specification formalisms.

The paper is structured as follows. In Section 2 we recall the idea of program development by stepwise refinement and transformation, and advocate theorem provers as a possible framework for such developments. Section 3 presents a method of theorem abstraction, which is at the core of our approach to generalise developments. Section 4 reviews the basic notions of data refinement and simulation, and shows how an example of data refinement can be treated in the setting of a theorem prover. Section 5 applies the abstraction method to the example in order to derive a generalised transformation rule. Finally, we compare to related work in Section 6 and conclude in Section 7.

2. Refinement by transformation

This section reviews the basic principles of stepwise refinement by transformation and how transformation can be modelled within a theorem prover. A formal *specification* of a program or system is an abstraction on the actual system, designed to make its properties clear and reasoning about them feasible. In a formal *development* method, the specification formalism comes equipped with at least one transitive refinement relation $S' \sqsubseteq S$ which expresses that a specification S' is a correct specialisation of another specification S. Refinement is used to verify that a proposed system design preserves the properties of its specification, i.e. the correctness of the implementation. However, as the program may be substantially different from the original specification, it is preferable to repeatedly prove refinement for minor changes one at a time until one has arrived at the final program. A *stepwise refinement process* can be understood as a sequence of specifications

$$S_1, S_2, \dots, S_n, \tag{2.1}$$

in which each successive stage of the program development process is a correct refinement of its preceding stage, e.g. $S_{i+1} \sqsubseteq S_i$ holds for $i = 1, \dots, n-1$. The transitivity of the refinement relation guarantees that S_n is a refinement of S_1. Usually, S_1 is an initial (abstract) specification and S_n is a program (or concrete specification), but various cases in between may be considered; for example, refining a requirement specification to a design specification or refining an executable specification to an equivalent but more efficient one. This incremental approach has been advocated and studied for different notions of refinement in the literature [Hoare 1972, Broy 1997, Back and von Wright 1998, de Roever and Engelhardt 1998, Sannella 2000]. However, refinement relations provide little guidance for program development. Stepwise refinement is essentially an "invent and verify" process to prove the correctness of development steps a posteriori; stepwise refinement is not an effective method for program development.

2.1 Transformational development

Transformational development offers a solution to this problem. Instead of inventing refinement steps, each step arises from applying a *transformation rule*, which has previously been proved correct. Thus, transformation rules may serve to guide the development and to reduce the amount of proof work.

In general, a transformation rule on terms consists of *parameters* $p_1, \dots, p_n$, an *applicability condition* A, an *input pattern* I, and an *output pattern* O:

$$\forall p_1, \dots, p_n \cdot A \Longrightarrow I \rightsquigarrow O \tag{2.2}$$

A transformation $I \rightsquigarrow O$ preserves refinement if $O \sqsubseteq I$. A transformation rule such as (2.2) is *correct* if the theorem $\forall p_1, \ldots, p_n \cdot A \Longrightarrow O \sqsubseteq I$ holds.

When a correct transformation rule is applied to a specification (or term), the applicability condition of the rule must be proved. In order to apply a transformation rule such as (2.2) to a term t, the input pattern I must match a subterm of t, say t_0, so that $t = C[t_0]$ where $C[\,]$ is the *context*. Let σ be a substitution which appropriately instantiates I, i.e. $I\sigma = t_0$. Then $I\sigma$ may be replaced by $O\sigma$ at the position of this subterm, i.e. the current specification $t = C[t_0] = C[I\sigma]$ is transformed to $C[O\sigma]$. Parameters which occur in the output pattern O but not in the input pattern I will not be instantiated by this match; their instantiation is left to the user. The instantiated applicability condition $A\sigma$ becomes a *proof obligation* which ensures the correctness of the transformational development. When $A\sigma$ holds, we know that $C[O\sigma] \sqsubseteq C[I\sigma]$. A stepwise development of S_1 to S_m is obtained by applying transformation rules $R_1, \ldots, R_{m-1}$ to show

$$S_1 \sqsupseteq S_2 \sqsupseteq S_3 \sqsupseteq \ldots \sqsupseteq S_m$$

and transitivity is used to deduce the refinement $S_1 \sqsupseteq S_m$.

For reflexive refinement relations, equality preserves refinement. For any equation $A = B$, Burstall and Darlington [1977] have shown how to derive and use two transformation rules

$$\forall p_1, \ldots, p_n \cdot A = B \Longrightarrow B \rightsquigarrow A \qquad (UNFOLD)$$
$$\forall p_1, \ldots, p_n \cdot A = B \Longrightarrow A \rightsquigarrow B \qquad (FOLD)$$

where $p_1, \ldots, p_n$ are the variables occurring free in A or B. Folding and unfolding rules are typically used for function definitions, axioms, and α-conversion. Some transformation rules are just standard rules of logic, while others are complex *design rules*, e.g. implementing a function specification by tail recursion using a divide and conquer rule [Smith 1985]. For design rules, application cannot be automated, so a system or tool for stepwise refinement by transformation can never be fully automatic.

2.2 Transformation systems and theorem provers

Many transformation systems such as CIP [Bauer *et al.* 1985], KIDS [Smith 1990], KIV [Reif *et al.* 1998, Reif and Stenzel 1993], PROSPECTRA [Hoffmann and Krieg-Brückner 1993], and Specware [Smith 1999] have been constructed from scratch, with a built-in notion of correctness, a fixed notion of refinement, and a given library of transformation rules. However, transformation systems can profitably be encoded in general purpose theorem provers. The theorem prover helps organise the overall development and provides proof support for discharge of applicability conditions. If the theorem prover itself is correct, and every transformation rule has been proved correct inside the theorem prover, correctness of the overall development is guaranteed. This approach has particularly been investigated for the Refinement Calculus of Back and von Wright [1998]; examples are found in the work of Staples [1998] and Hemer *et al.* [2001], and with the Refinement Calculator [Långbacka *et al.* 1995, Butler *et al.* 1997]. Program transformation based on rewriting program schemas and a second-order matching algorithm was first proposed by Huet and Lang [1978]. Recent examples of transformation systems, implemented in the Isabelle prover, can be found in e.g. Anderson and Basin [2000], Hemer *et al.* [2001], and Lüth and Wolff [2000].

In the theorem prover approach, a transformation rule of the form (2.2) can be given by a *theorem*

$$A \Longrightarrow I \sqsupseteq O, \tag{2.3}$$

ranging over free variables. In order to reflect the correctness of arbitrary design choices, a standard refinement relation $\sqsubseteq$ is considered here, which is assumed to be reflexive, transitive, and monotone. Monotonicity can be treated in a more fine-grained manner by means of window inference, cf. Back *et al.* [1997] and Grundy [1996], but in order to keep the exposition simple these issues shall be ignored here.

Design transformations such as those mentioned above have been derived by a careful generalisation of known examples. We propose to formalise this process within the formal development system. In the transformational approach, every development step amounts to applying a particular transformation rule. Consequently, transformation rules are schematic rules in a deduction system and transformational program development amounts to proving theorems about refinement. On the other hand, every proof of a refinement such as (2.1) gives rise to a transformation rule. More applicable rules can be obtained by generalising previous developments, which amounts to generalising theorems.

3. Generalising theorems

This section presents a method for abstracting theorems by means of proof transformations. Let π be a proof of a theorem ϕ. The generalisation strategy will transform π into a proof of a schematic theorem in a stepwise manner. The transformation process consists of three phases:

(1) making assumptions explicit;

(2) abstracting function symbols;

(3) abstracting types.

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 a schematic theorem $\psi \Longrightarrow \phi'$ is derived, where ϕ' is a modification of the initial formula ϕ. In such theorems, the formulas of ψ are called *applicability conditions*.

3.1 Logical frameworks

The proposed abstraction technique consists of transforming theorems in a given logic into derived inference rules of the same logic. This can be done inside the logical language if a logical framework style theorem prover is used. A logical framework is, roughly speaking, a meta-level inference system which can be used to specify different object-level deductive systems. This section presents some key ideas about logical frameworks that underlie our abstraction process. The formal foundation for logical frameworks was proposed by Harper *et al.* [1993], for a recent overview see Pfenning [2001]. The present work uses the Isabelle prover [Paulson 1990].

In order to represent an object logic in a logical framework (or meta-logic), an abstract syntax is defined for the object logic, encoding its formulas as meta-logical terms. Derivability in the object logic is represented by a predicate on the terms of the meta-logic, which is defined by axioms encoding the inference rules and axioms of the object logic. For example, each logical symbol of a natural deduction system will have associated introduction and elimination axioms. Implication elimination (modus ponens) is represented in Isabelle's meta-logic by the higher-order axiom $[\![\mathbf{a} \Longrightarrow \mathbf{b};\ \mathbf{a}]\!] \Longrightarrow \mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are meta-variables, i.e. placeholders for logical formula in the object logic, and ';' and '$\Longrightarrow$' denote conjunction and implication at the meta-level. Meta-level implication reflects object level derivability.

Logical frameworks such as Isabelle use higher-order resolution between formulas at the meta-level [Paulson 1990] to modify proof states. The proof state consists of a rule, the premises of which are the remaining subgoals. If a rule's conclusion can be unified with a subgoal, the subgoal is replaced by the instantiated premises of the rule; if the rule has no premises, the subgoal is removed. The proof of a formula ϕ starts with a proof state $[\![\phi]\!] \Longrightarrow \phi$. When no subgoals remain, the formula ϕ has been proved.

The logical framework allows proof of theorems directly in the meta-logic. Theorems of the meta-level containing meta-variables are referred to as *schematic theorems*. The correctness of all instantiations of a proved schema follows from the correctness of the representation of the rules of the object logic. Schematic theorems about meta-logic implication are derived inference rules of the object logic. In logical frameworks, new object logic inference rules can therefore be derived within the logical language.

We shall now describe the generalisation of theorems as sketched above. The work presented here uses Isabelle, but the general principle holds for other logical frameworks. For the (non-trivial) details of the implementation in Isabelle, the reader is referred to Johnsen and Lüth [2003]. Note that a necessary condition for the second abstraction step is that the logical framework permits higher-order variables, and for the third step that the logical framework permits type variables.

3.2 Making proof assumptions explicit

In theorem provers, a theorem is generally proved in a theory which includes a context of axioms and previously established theorems. In a theorem prover with automated proof support, it is not always known which theorems have been used to solve a proof goal. However, if the proof of the theorem can be reconstructed, these can be retrieved. In a natural deduction proof, these theorems can be introduced at the leafs of the proof tree. In order to make a contextual theorem explicit, the branch is closed by applying the implication introduction rule.

To illustrate this, consider the proof π of theorem ϕ. At the leaf of a branch π_i in the proof, a theorem or definition needed in the proof is found, say ψ_i. The branch π_i may be closed by applying $\Longrightarrow$-introduction at the root of the proof, which leads to a proof of the formula $\psi_i \Longrightarrow \phi$. This process is repeated for every branch in π with a relevant leaf theorem. If we need to make j theorems explicit, we derive a proof π' of the formula

$$(\psi_1 \wedge \ldots \wedge \psi_j) \Longrightarrow \phi. \tag{3.1}$$

If the applicability conditions in the proof of a leaf formula ψ_i of the proof π are included among the other leaf formulas of π, π can be expanded with the proof of ψ_i at the appropriate leafs to remove superfluous applicability conditions from (3.1) before the proof transformation. Also, a formula ψ_i must be transformed into a closed formula by quantifying over all free variables before it is made explicit.

It is possible to improve the method presented above by weakening the applicability conditions of the derived theorem. This can be done by repeatedly removing a leaf which is followed by an elimination rule in the proof tree before applying implication introduction to the leaf formula. For example if $\forall$-elimination is considered, the obtained applicability condition will be an instance of the original leaf theorem, rather than the theorem itself.

3.3 Abstracting function symbols

The next phase of the transformation process consists of replacing function symbols by variables. When all implicit assumptions (axioms and theorems) concerning a function symbol have been made explicit, as in (3.1) above, all the information about this function symbol that is needed in the proof, is contained within the derived theorem. The function symbol has become an *eigenvariable* because the proof of the theorem no longer depends on the theorem prover context with regard to this function symbol. Therefore, such function symbols can be replaced by variables throughout the proof. Let $\phi[t_1/t_2]$ denote a formula ϕ and $\pi[t_1/t_2]$ a proof π after substitution, where the term or variable t_1 has been replaced by the term or variable t_2 in ϕ and π, renaming bound variables as needed to avoid variable capture.

A central idea in logical framework encodings is to represent object logic variables directly by meta-logic variables [Pfenning 2001]. Hereafter, all free variables will be meta-variables. Consequently, the abstraction process replaces function symbols by meta-variables. For a function symbol F in theorem (3.1), one may therefore derive a new theorem

$$(\psi_1(t_1^1, \ldots, t_n^1)[F/\mathbf{a}] \wedge \ldots \wedge \psi_i(t_1^j, \ldots, t_m^j)[F/\mathbf{a}]) \Longrightarrow \phi[F/\mathbf{a}], \tag{3.2}$$

where $\mathbf{a}$ is a meta-variable, i.e. a placeholder for object logic terms, by transforming the proof π' into a new proof $\pi'[F/\mathbf{a}]$.

EXAMPLE 1. (CONCRETE CALCULATIONS) As a simple example, consider as object logic a higher-order equational logic, with axioms including transitivity (*trans* $\equiv [\![\mathbf{a} = \mathbf{b};\ \mathbf{b} = \mathbf{c}]\!] \Longrightarrow \mathbf{a} = \mathbf{c}$), congruence (*arg_cong* $\equiv [\![\mathbf{x} = \mathbf{y}]\!] \Longrightarrow \mathbf{f}\ \mathbf{x} = \mathbf{f}\ \mathbf{y}$), symmetry (*sym*), reflexivity (*refl*), etc. In this object logic, consider a theory including the standard operations 0, S (successor), +, and the defining axioms for addition on the type $\mathbb{N}$ of natural numbers:

$$\text{Ax1: } \forall x \cdot x + 0 = x \qquad \text{Ax2: } \forall x, y \cdot x + Sy = S(x + y)$$

In this theory, the theorem $S0 + S0 = SS0$ can be proved as follows:

$$\dfrac{\dfrac{\text{Ax2}}{S0 + S0 = S(S0 + 0)} \quad \dfrac{\dfrac{\text{Ax1}}{S0 + 0 = S0}}{S(S0 + 0) = SS0}\,\textit{arg\_cong}}{S0 + S0 = SS0}\,\textit{trans} \tag{3.3}$$

Following the procedure, the proof is transformed into a proof π of

$$(S0 + S0 = S(S0 + 0) \wedge S0 + 0 = S0) \Longrightarrow S0 + S0 = SS0,$$

removing the axioms from the leafs of proof tree and closing the branches. Next the function symbol 0 is abstracted, resulting in a derived theorem

$$(S\mathbf{a} + S\mathbf{a} = S(S\mathbf{a} + \mathbf{a}) \wedge S\mathbf{a} + \mathbf{a} = S\mathbf{a}) \Longrightarrow S\mathbf{a} + S\mathbf{a} = SS\mathbf{a},$$

where the new proof π' is given by $\pi[0/\mathbf{a}]$. (This is allowed since 0 in the proof no longer depends on the context.) As a next step, the function symbol S is abstracted, resulting in the theorem

$$\begin{aligned}(\mathbf{b}(\mathbf{a}) + \mathbf{b}(\mathbf{a}) = \mathbf{b}(\mathbf{b}(\mathbf{a}) + \mathbf{a}) \wedge \mathbf{b}(\mathbf{a}) + \mathbf{a} = \mathbf{b}(\mathbf{a}))\\ \Longrightarrow \mathbf{b}(\mathbf{a}) + \mathbf{b}(\mathbf{a}) = \mathbf{b}(\mathbf{b}(\mathbf{a})),\end{aligned}$$

from the transformed proof $\pi'[S/\mathbf{b}]$. Finally, the infix function symbol '+' is abstracted (replaced by the prefix meta-variable $\mathbf{c}$), yielding the theorem

$$\begin{aligned}\mathbf{c}(\mathbf{b}(\mathbf{a}), \mathbf{b}(\mathbf{a})) = \mathbf{b}(\mathbf{c}(\mathbf{b}(\mathbf{a}), \mathbf{a})) \wedge \mathbf{c}(\mathbf{b}(\mathbf{a}), \mathbf{a}) = \mathbf{b}(\mathbf{a}))\\ \Longrightarrow \mathbf{c}(\mathbf{b}(\mathbf{a}), \mathbf{b}(\mathbf{a})) = \mathbf{b}(\mathbf{b}(\mathbf{a})),\end{aligned}$$

by repeating the procedure. In the logical framework setting, a schematic (inference) rule has been derived, which allows any term $\mathbf{c}(\mathbf{b}(\mathbf{a}), \mathbf{b}(\mathbf{a}))$ of type $\mathbb{N}$ to be replaced by $\mathbf{b}(\mathbf{b}(\mathbf{a}))$, provided that the applicability conditions $\mathbf{c}(\mathbf{b}(\mathbf{a}), \mathbf{b}(\mathbf{a})) = \mathbf{b}(\mathbf{c}(\mathbf{b}(\mathbf{a}), \mathbf{a}))$ and $\mathbf{c}(\mathbf{b}(\mathbf{a}), \mathbf{a}) = \mathbf{b}(\mathbf{a}))$ hold.

3.4 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, type constants can now be replaced by type variables. The higher-order resolution mechanism of the theorem prover will instantiate the type variables as well as the term variables when the inference rule is applied. If Example 1 is considered, this works well, yielding an inference rule which can be used to prove theorems about other types than $\mathbb{N}$. However, the formal languages used by theorem provers have structured types which may give rise to *type-specific inference rules*, such as induction. When these occur in the proofs, they must also be made explicit before type abstraction can succeed. This is illustrated by the following example.

EXAMPLE 2. (PROOF BY INDUCTION) Consider proving $x + 0 = 0 + x$, using the object logic and theory of Example 1, extended with the schema for natural induction:

$$\frac{\mathbf{P}0 \quad \forall\, t \cdot \mathbf{P}t \Longrightarrow \mathbf{P}(St)}{\mathbf{P}x}\ \mathit{ind} \tag{3.4}$$

The higher-order meta-variable $\mathbf{P}$ allows a formulation of the induction schema as a single rule. The following proof of the theorem is slightly edited for brevity:

$$\dfrac{\dfrac{}{0+0=0+0}\,\mathit{refl} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix}\text{Ax1}\\ \vdots\end{matrix}}{St+0=S(t+0)} \qquad \dfrac{\begin{matrix}[t+0=0+t]^1\\ \vdots\end{matrix} \qquad \begin{matrix}\text{Ax2}\\ \vdots\end{matrix}}{S(t+0)=0+St}\,\mathit{trans}}{St+0=0+St}\,\mathit{trans}}{t+0=0+t \Longrightarrow St+0=0+St}\,{\Longrightarrow}\text{-}\mathit{intro}_1}{\forall\, t \cdot t+0=0+t \Longrightarrow St+0=0+St}\,\forall\text{-}\mathit{intro}}{x+0=0+x}\,\mathit{ind} \tag{3.5}$$

For abstraction over the type $\mathbb{N}$ of natural numbers here, the induction schema must be made explicit as an assumption in the derived theorem. In the logical framework, induction is treated as a meta-logic theorem and not as the object logic inference rule of (3.5). Let *ind'* denote the theorem corresponding to the instantiated induction rule:

$$\begin{aligned} \mathit{ind'} &\stackrel{\mathit{def}}{=} \mathit{ind}[\mathbf{P}/(\lambda\, x \cdot x + 0 = 0 + x)] \\ &= \quad (0 + 0 = 0 + 0 \wedge (\forall\, t \cdot t + 0 = 0 + t \Longrightarrow St + 0 = 0 + St)) \\ &\qquad \Longrightarrow x + 0 = 0 + x. \end{aligned}$$

Furthermore, let π_1 and π_2 denote the two proof trees above the induction rule in Proof (3.5), which can now be replaced by the following proof:

$$\dfrac{\dfrac{\dfrac{\pi_1}{0+0=0+0} \quad \dfrac{\pi_2}{\forall\, t \cdot t+0=0+t \Longrightarrow St+0=0+St}}{0+0=0+0 \wedge \forall\, t \cdot t+0=0+t \Longrightarrow St+0=0+St}\,\wedge\text{-}\mathit{intro} \qquad \mathit{ind'}}{x+0=0+x}\,{\Longrightarrow}\text{-}\mathit{elim} \tag{3.6}$$

Following the outlined method, a proof may here be derived for the theorem

$$ind' \wedge (S(0 + t) = 0 + St) \wedge (t + 0 = t) \wedge (St + 0 = St) \Longrightarrow x + 0 = 0 + x.$$

In order to abstract over the type constant $\mathbb{N}$, the function symbols from the theory of natural numbers must be abstracted. We replace 0, S, and + by meta-variables **a**, **b**, and **c**, respectively. Then, the type constant $\mathbb{N}$ can be replaced by a type variable, resulting in a proof for the theorem

$$\begin{aligned}&(ind')[0/\mathbf{a}, S/\mathbf{b}, +/\mathbf{c}]\\&\quad \wedge \mathbf{b}(\mathbf{c}(\mathbf{a}, t)) = \mathbf{c}(\mathbf{a}, \mathbf{b}(t)) \wedge \mathbf{c}(t, \mathbf{a}) = t \wedge \mathbf{c}(\mathbf{b}(t), \mathbf{a}) = \mathbf{b}(t)\\&\qquad\qquad \Longrightarrow \mathbf{c}(\mathbf{x}, \mathbf{a}) = \mathbf{c}(\mathbf{a}, \mathbf{x}),\end{aligned}$$

which can be applied as an inference rule to any type. To discharge the applicability conditions when the rule is applied, the formula corresponding to the instantiated induction rule must be a theorem for the new type.

The induction rule in Example 2 illustrates a more general problem: in logical systems where new datatypes may be defined, there will be proof rules that are specific to each datatype, such as structural induction, rules for various recursion operators, and their duals (coinduction, corecursion). In our setting, type constructors, induction rules, and recursive function definitions result in axioms that may need to be made explicit in the transformation process before type abstraction can be applied.

Abstraction over types demonstrates the advantage of working with abstraction in logical frameworks: inference rules are represented by meta-logic theorems and object logic types by meta-logic terms, so type-specific inference rules and type constants can be handled by the abstraction process in the same way as object logic theorems and function symbols. Consequently, the result of every abstraction step remains within the logical language and every derived proof may be verified in the prover. The derived inference rules may be applied to formulas of any type defined in the object logic.

Different type-specific inference rules are reflected by different schematic theorems, depending on the number and profiles of the type constructors. It is this information which is preserved when the type-specific rules are made explicit as applicability conditions for the theorem during the abstraction process. Hence, to discharge the derived applicability condition, the target type must have the same or stronger type-specific inference rules.

For the abstraction process, it does not matter whether the logical framework constructs datatypes conservatively (like Isabelle, where new datatypes are encoded in a previously defined representing universe [Berghofer and Wenzel 1999, Paulson 1997]) or axiomatically by asserting relevant axioms, since theorems and axioms are treated uniformly in the meta-logic. The abstraction is done at the meta-level and the derived theorems apply to any type for which the corresponding instantiation of the induction rule holds.

3.5 Abstractions tactics

The preceding sections have shown how theorems may be generalised by abstracting over assumptions, function symbols, and types. More advanced *abstraction tactics*, which automate the derivation of some inference rules, can be built by combining these basic abstraction steps.

Isabelle is organised in *theories*. A theory can be perceived as a signature, defining types, operations, and (rarely) axioms. Every theorem belongs to the theory in which it is proved. Theories are organised hierarchically, so all theorems from (direct and indirect) ancestor theories remain valid in a theory. Hence, a useful abstraction tactic in the setting of Isabelle is to abstract a theorem from a theory T_1 to an ancestor theory T_2. This corresponds to abstracting over all additional definitions and theorems that T_1 provides over T_2. The tactic gathers all theorems, operations, and types from the proof which do not occur in T_2 and calls the basic abstraction steps recursively, starting with theorems and continuing with function symbols and types. Theorems may thus be moved from one branch of the theory tree to another by abstraction to a common ancestor theory.

Another useful tactic is to generalise a specific type in a theorem. Before this is feasible, all operations using this type and all theorems referring to any of these operations need to be abstracted. Thus, the tactic works its way up from a given type, identifying all operations and theorems depending on the given type, abstracting these and then abstracting the type itself. An example of this tactic is found in Section 4.4.

4. Data refinement and simulation

The approach is illustrated by abstraction in the context of *data refinement*. Data refinement captures the idea that a concrete representation of a data structure preserves the correctness of the reasoning done for an abstract representation of the data structure. Data refinement was originally introduced by Hoare [1972], based on concepts from Simula 67 [Dahl *et al.* 1968], and has been studied in various specification languages; for overviews see the books of de Roever and Engelhardt [1998] or Derrick and Boiten [2001].

We distinguish between a user-defined *data structure* (see Definition 1 below) and a *datatype*, which is a constructor for an axiomatic theory for a free structure, as provided by theorem provers. In contrast to data structures, theorem prover datatypes come equipped with inference rules (see Section 3.4). Data refinement is here presented in terms of relations on state spaces. Let $X \leftrightarrow Y$ denote a relation between sets X and Y.

4.1 Refinement of data structures

This section defines a data structure, and informally introduces data refinement. Let *Var* and *Val* denote the types of variable names and data values, respectively. A *state* is a partial function $\mathit{Var} \rightarrow\!\!\!\!\!+ \mathit{Val}$. A data structure comprises a local state of values and an indexed collection of operations; the indexing set can be thought of as a signature. Operations may read and write to a global state, denoted G. Before a data structure may be used by a program, the local state variables must be initialised. The local state space of a data structure can *only* be manipulated through its operations.

DEFINITION 1. (DATA STRUCTURE) *A* data structure *is a tuple* $\mathcal{X} = \langle X, xi, \{xop_i\}_{i \in I} \rangle$, *where*

(1) X *is the* local state space*,*

(2) $xi \in G \leftrightarrow X$ *is the* initialisation *relation,*

(3) $\{xop_i\}_{i \in I}$ *is an indexed collection of relations in* $X \times G \leftrightarrow X \times G$*, the* operations*.*

Calligraphic letters $\mathcal{X}$, $\mathcal{A}$, and $\mathcal{C}$ denote data structures. The initialisation relation xi is total, but operations xop_i may be partial. In practice, the local state space X will often be further restricted by a type invariant. A *program* $P(\mathcal{X})$ manipulating a data structure $\mathcal{X}$, is a sequence of operations on $\mathcal{X}$ starting with an initialisation. Due to the encapsulation of the local state, the program is parametric over all data structures with the same index set. Let $\mathcal{A}$ and $\mathcal{C}$ be two data structures, so $P(\mathcal{C})$ is obtained from $P(\mathcal{A})$ by replacing the operations of $\mathcal{A}$ by those of $\mathcal{C}$. $\mathcal{C}$ *refines* $\mathcal{A}$ if, for every program P, the global state after executions of $P(\mathcal{C})$ and $P(\mathcal{A})$ are the same, expressed by relational inclusion as $P(\mathcal{C}) \subseteq P(\mathcal{A})$. Formal definitions and discussions of different formulations of data refinement may be found in the overview books [de Roever and Engelhardt 1998, Derrick and Boiten 2001].

4.2 Simulation

To prove that a refinement relation holds between two data structures involves quantifying over *all* possible programs that may use the data structures. Hence, the definition of data refinement does not embody an "effective method for proving data refinement" [de Roever and Engelhardt 1998]. Therefore, data refinement is proved via *simulation*, comparing the effect of the data structures' operations on the local state by a retrieve relation. Downward simulation, which is most commonly used, is considered here.

The possible partiality of data structure operations complicates simulation. Partiality is usually handled by totalising state spaces and lifting the retrieve relation to totalised states. Instead, partiality is treated here by considering the domain and range of operations explicitly. The equivalence of these approaches has been shown by Woodcock and Davies [1996]. If x is a state in the state space X and $R : X \leftrightarrow X$ is a relation between states, x' denotes a state such that $(x, x') \in R$ (for a given state x and relation R). The *precondition* or *domain* of a (partial) operation is defined as a set of states:

DEFINITION 2. (DOMAIN) *Let X be a state and op an operation $X \leftrightarrow X$. The* domain *of op is the set*

$$\operatorname{dom} op \stackrel{def}{=} \{x \mid \exists x' \cdot (x, x') \in op\}.$$

Adapting a result of Woodcock and Davies [1996], downward simulation is defined directly as proof conditions on the state spaces.

DEFINITION 3. (DOWNWARD SIMULATION) *Let G be a global state, I an index set, and $\mathcal{A} = \langle A, ai, \{aop_i\}_{i \in I}\rangle$ and $C = \langle C, ci, \{cop_i\}_{i \in I}\rangle$ data structures. C simulates $\mathcal{A}$ if there is a retrieve relation $R : A \leftrightarrow C$ between the local states, such that the following conditions hold:*

$$
\begin{array}{ll}
PC_1 & \forall\, c' \in C, g \in G \cdot (g, c') \in ci \Longrightarrow \exists\, a' \in A \cdot (g, a') \in ai \wedge (a', c') \in R \\
PC_2 & \forall\, i \in I\, \forall\, a \in A, c \in C, g \in G \cdot \\
 & \quad (a, g) \in \operatorname{dom} aop_i \wedge (a, c) \in R \Longrightarrow (c, g) \in \operatorname{dom} cop_i \\
PC_3 & \forall\, i \in I\, \forall\, a \in A, c, c' \in C, g, g' \in G \cdot \\
 & \quad (a, g) \in \operatorname{dom} aop_i \wedge (a, c) \in R \wedge ((c, g), (c', g')) \in cop_i \\
 & \qquad \Longrightarrow \exists\, a' \in A \cdot ((a, g), (a', g')) \in aop_i \wedge (a', c') \in R
\end{array}
$$

The conditions are referred to as initialisation, applicability, and correctness.

4.3 Simulation as transformation

Downward simulation is sound but not complete with respect to refinement, so simulation entails refinement. Soundness can be expressed as a theorem

$$\forall\, \mathcal{A}, C, R \cdot PC_1 \wedge PC_2 \wedge PC_3 \Longrightarrow \mathcal{A} \sqsupseteq C, \tag{4.1}$$

which becomes a transformation rule (see Section 2.1 above). The input pattern of the rule mentions $\mathcal{A}$, but neither C nor R, which are parameters of the transformation rule, to be supplied by the user when the transformation rule is applied. Although more specific than refinement, the simulation relation is still too general to be of practical assistance for making design decisions; in particular, it does not identify the concrete data structure C or the retrieve relation R. Simulation provides an effective proof method for verifying refinement steps, but it does not make a good transformation rule.

4.4 An example of data refinement

Data refinement is illustrated by stacks, which may be implemented either by lists or by an array with a pointer, inspired by an example from de Roever and Engelhardt [1998]. The stack has four operations; the empty stack corresponds to the initialisation relation, and the index set comprises the other operations $I = \{push, pop, top\}$. An operation $f : X \leftrightarrow X$ relates state variables $x \in X$ before and after operation application. Formally, $f \equiv E(x, x')$ describes the relation $f \stackrel{def}{=} \{(x, x') \mid E(x, x')\}$, where E is a predicate on state variables.

4.4.1 The stack as a list

A stack over the integers $\mathbb{Z}$ can be specified using finite lists. Assume a predefined data type $List[T]$ with constructor *cons* and selectors *hd* and *tl*; the empty list is denoted $[\,]$. For any list l, $len(l)$ denotes the number of elements in l and $l!n$

(for $0 \leq n < len(l)$) the n'th element of l. The mapping of I to the stack operations is evident.

data structure ListStack $\equiv$
global $(i : \mathbb{Z}, o : \mathbb{Z})$
local $l : List[\mathbb{Z}]$
init $lempty \equiv \{((i, o), l) \mid l = [\,]\}$
ops $lpush \equiv i = i' \land o = o' \land l' = Cons(i, l)$
$lpop \equiv l \neq [\,] \land i = i' \land o = o' \land l' = tl(l)$
$ltop \equiv l \neq [\,] \land i' = i \land o' = hd(l) \land l' = l$

4.4.2 *The stack as an array with a pointer*

The stack may be implemented by an array and a variable p: the content is stored in the array and p points to the next free entry. Assume a predefined data type $Array[T]$ of arrays indexed by natural numbers. If a is a sufficiently large array of type T, $t \in T$, and $n \in \mathbb{N}$, then $a[n := t]$ is the array obtained by updating a with t at index n. Let *empty* be the empty array.

data structure ArrayStack $\equiv$
global $(i : \mathbb{Z}, o : \mathbb{Z})$
local $(l : Array[\mathbb{Z}], p : \mathbb{N})$
init $rempty \equiv \{((i, o), (l, p)) \mid l = empty, p = 0\}$
ops $rpush \equiv i' = i \land o' = o \land p' = p + 1 \land a' = a[p := i]$
$rpop \equiv i' = i \land o' = o \land p' = p - 1 \land a' = a$
$rtop \equiv i' = i \land o' = a[p - 1] \land a' = a \land p' = p$

Initialisation of *ArrayStack* sets the stack pointer p to 0. As pointed out by de Roever and Engelhardt [1998], the specifications are not equal. In particular, *ArrayStack* does not satisfy the equation $pop(push(t, s)) = s$, which holds for *ListStack*. However, the stacks cannot be distinguished by the observer functions on the data structures.

4.5 *Proving simulation*

Proof that *ArrayStack* refines *ListStack* is via simulation. A retrieve relation from the state space of *ArrayStack* to the state space of *ListStack* is needed, relating the elements of the list in *ListStack* to elements of the array in *ArrayStack*. While the array grows at the back, the list grows at the front; therefore, the top of *ArrayStack* is always $a[p - 1]$ if p is the stack pointer index, whereas the top of *ListStack* is $hd(a)$ or $a!0$. This suggests the following definition of a retrieve relation $Ret : A \times C$ between abstract states $A = List[\mathbb{Z}]$ and concrete states $C = \mathbb{N} \times Array[\mathbb{Z}]$:

$$Ret \equiv \{(l, (p, a)) \mid \; p = len(l) \land \forall i \in \mathbb{N} \cdot 0 \leq i < p \Longrightarrow l!i = a[p - i - 1]\}$$

Using Theorem (4.1), *ListStack* is refined to *ArrayStack* by instantiating $\mathcal{A}$ with *ListStack*, C with *ArrayStack*, and R with *Ret*. The resulting instantiated proof obligations PC_1, PC_2, and PC_3 are discharged using Isabelle.

4.6 Modelling the data refinement in Isabelle

The preceding definitions are formalised naturally in Isabelle/HOL and organised in Isabelle theories `DStruct.thy`, `DataRef.thy`, `ListStack.thy`, `ArrayStack.thy`, and `StackRef.thy`, built hierarchically on each other. The formalisation is now briefly sketched. Let `i`= {*push*, *pop*, *top*} be an enumeration type of indices. A data structure is represented by a tuple parametrised over global and local states, which in Isabelle syntax becomes

```
types ('g, 'a) dstruct =
      "('g* 'a) set* (i=> (('g* 'a)* ('g* 'a)) set)"
```

Let the type `globalstate` be a product of integers which captures input and output variables, and `lstate` a list of integers which captures the local state, using Isabelle/HOL's predefined type `list`. Finally, the *ListStack* type `lstack` is defined as `(globalstate, lstate) dstruct`, instantiating the data structure with appropriate global and local states. The operation definitions are now straightforward. For example, `lpush` becomes

```
lpush :: "((globalstate* lstate)* (globalstate* lstate)) set"
         "lpush == {(((i, out), l), ((i', out'), l')).
                     (i', out')= (i, out) & (l'= i # l)}"
```

The indexed set `lops` of operations is defined by a case distinction over `i`, and *ListStack* as an element `ls == (lempty, lops)` of type `lstack`.

ArrayStack is implemented similarly, using Isabelle/HOL's `map` type. A map is a partial function from domain `'a` to range `'b`, written `'a ~=> 'b`. Arrays are partial functions with the natural numbers as domain.

For this example, refinement is defined axiomatically as simulation. Let constants `pc1`, `pc2`, and `pc3` abbreviate the three proof obligations. Data refinement relates data structures with the same global state space:

```
refines :: "[('g, 'a) dstruct, ('g, 'b) dstruct] => bool"
           "c [= a == EX r. let cinit= fst c; cops= snd c;
                                ainit= fst a; aops= snd a
                            in  (pc1 r ainit cinit &
                                (!i. pc2 r (aops i) (cops i))&
                                (!i. pc3 r (aops i) (cops i)))"
```

Consequently, `c` refines `a` if there is a retrieve relation `r` satisfying the proof obligations. A more elegant approach would be to define refinement by inclusions between the initialisation and operation relations, and prove that simulation is sound with respect to this relation. However, this requires a formalisation of the soundness proof of simulation inside Isabelle/HOL, an interesting exercise in its own right but out of the scope of the present paper.

Refinement is shown by providing a retrieve relation between `ArrayStack` and `ListStack` for the existentially bound `r` in the definition above. The instantiated proof obligations `pc1`, `pc2`, and `pc3` remain to be proved. While `pc1` is easily discharged by Isabelle, the others require some effort. The proof obligations resulting from `pc3` are harder to show than those resulting from `pc2`; an invariant relating the contents of the array and the list is needed.

5. Transformation rules for data refinement

Although effective for verifying the correctness of development steps, the simulation rule does not provide an effective method for development of specifications. The development process still follows the invent and verify strategy: for each step the developer must provide a new data structure, a retrieve relation, and a correctness proof. Transformational development cannot replace this strategy altogether, but the strategy can be incrementally supplemented by generalising and reusing previously established developments.

In this section, the abstraction process outlined in Section 3 is applied to the refinement proof from Section 4.4. The goal of the abstraction process here is not to derive rules which are as general as possible, but to derive constructive and easily applicable rules; given a data structure, the prover's instantiation mechanism for rule application provides the target data structure, and the retrieve relation. Correctness of the transformation follows by instantiation from the correctness proof of the transformation rule. If applicability conditions need to be verified, these should be easier than to prove than simulation directly. When applicable, derived rules may provide constructive development guidelines and thus supplement the invent and verify strategy, which is still available when the transformation rules do not apply.

5.1 Basic transformation rules

The derivation of constructive transformation rules is essentially a bottom-up process, generalising examples of refinements into logical rules. A transformation rule was derived and its correctness proved in Section 4.5: the example refinement of Section 4.4 is an instance of the transformation rule for downward simulation (4.1), and the proof obligations correspond to the applicability conditions of the transformation rule. The proof obligations of the example have all been discharged, yielding a basic transformation rule

$$\mathit{ListStack} \sqsupseteq \mathit{ArrayStack} \tag{5.1}$$

without any applicability conditions. However, the derived rule is very specific; it can only be applied to this particular formalisation of *ListStack*. Hence, we want to make it applicable to more data structures.

5.2 Making definitions explicit

The first step in the abstraction process is to identify the definitions and lemmas on which the proof of the refinement theorem (5.1) depends, and make them explicit in the formulation of the theorem. In the implementation, this is done by applying abstraction tactics to the theorem (referred to as `stack_ref` below). In the example, there are three sets of definitions and lemmas to make explicit, corresponding to *ListStack*, *ArrayStack*, and the refinement between the two, respectively. These definitions are organised in different Isabelle theories. For readability the definitions related to the refinement are considered first, such as the retrieve relation, then the definitions of *ArrayStack*, and finally the definitions of *ListStack*. These definitions are identified by the function `Inspect.dep_in_thms`, given the name of the Isabelle theory where the refinement is defined and the name of the theorem:

```
ML> val Ref_deps = Inspect.dep_in_thms [StackRef.thy] stack_ref;
val Ref_deps = ["r_same_elems","r_def"] : string list
```

The result is here returned as a list `Ref_deps`, in which `r_def` is the name of the retrieve relation and `r_same_elems` is an auxiliary lemma used in the proof. An Isabelle tactic `AbsTac.abs_thms` takes a list of lemmas (and definitions) such as `Ref_deps` and the theorem `stack_ref`, and transforms the proof of `stack_ref` into the proof of a theorem where these lemmas occur explicitly, as described in Section 3.2. The derived theorem is named `R_thm`.

```
ML> val R_thm = AbsTac.abs_thms Ref_deps stack_ref;
val R_thm =
  "[| !!l k a. ((l, k, a) : r) = (l = elems k a);
      r == {(x, xa, xb). xa = length x &
       (ALL xc. 0 <= xc & xc < xa
          --> x ! xc = the (xb (xa - xc - 1)))} |]
   ==> EX x. let cinit = fst as; cops = snd as;
                 ainit = fst ls; aops = snd ls
          in (ALL (g, c):cinit. EX a. (g, a) : ainit & (a, c) : x) &
             (ALL i c. ALL (g, a):Domain (aops i).
                     (a, c) : x --> (g, c) : Domain (cops i)) &
             (ALL i c c' g'. ALL (g, a):Domain (aops i).
                     (a, c) : x & ((g, c), g', c') : cops i
                       --> (EX a'. ((g, a), g', a') : aops i
                                     & (a', c') : x))" : thm
```

In the applicability conditions of the derived rule, the first line is the lemma `r_same_elems` and the next three lines contain the definition of the retrieve relation. This process is repeated with the definitions and lemmas of *ArrayStack* and *ListStack*, resulting in a theorem `L_thm`, where all the definitions and auxiliary lemmas needed for the proof are explicit. The theorem is quite large, containing sixteen assumptions, and hence not included here. Note that at this point the transformation rule can still only be applied to the `ListStack` data structure, since the operations `aops` above are given by `snd ls` (which is `lops`), but the transformation rule can now be generalised.

5.3 Operation names in data structures

The next step in the generalisation process is to replace the function symbols of the example data structures by variables. This allows application of the transformation to `ListStack` data structures independent of the (arbitrary) names given to the stack operations and to the attributes of the data structure. The function `Inspect.dep_above_ops` identifies all function symbols introduced in Isabelle theories extending the data structure definition in `DStruct.thy`, i.e. in `ListStack`, `ArrayStack`, and the retrieve relation `r`.

```
ML> val dep_ops = Inspect.dep_above_ops DStruct.thy L_thm;
val dep_ops =
  ["r","as","aempty","aops","apush","apop","atop","elems",
   "ls","lempty","ltop","lops","lpush","lpop"] : string list
```

The order in which these symbols are abstracted is important. As the definitions of some operations may depend on others, abstraction must descend through the levels of this dependency graph. The abstraction tactic `AbsTac.rep_ops_thy` replaces function symbols with meta-logical variables as described in Section 3.3, one level at a time. Following Isabelle's convention of denoting meta-variables with a prefixed '`?`', `r` is replaced by the meta-variable `?r`, etc. In the resulting transformation rule `nofun_thm`, all the function symbols of the data structures have become variables that can be instantiated with any appropriately typed functions or variables during the development process.

5.4 A Polymorphic transformation rule for stacks

Recall from Section 3.4 that type constants in a transformation rule can be replaced by type variables when no function symbols of the rule belong to the type. This was achieved in the previous section, so the procedure can now be applied to the running example and the integer type is replaced by a type variable, by means of the `Abstract.abstract_type` abstraction tactic:

```
val abs_rule = Abstract.abstract_type
                (Type("integer", [])) ("a", ["HOL.type"]) nofun_thm;
```

In the resulting transformation rule `abs_rule`, the integer type has been replaced by a variable of type `HOL.type`, which can be instantiated by any type defined in Isabelle's higher-order logic. The derived transformation rule `abs_rule` can be applied to stacks of a variety of types by instantiating the meta-variable `?a` of the rule with the abstract data structure. The entire rule is included and explained in detail in Appendix A.

5.5 Reusing the transformation rule

Reuse of the derived transformation rule is now considered. Assume therefore a stack of strings, specified by a list as in Section 4.4:

data structure StringList ≡
global $(i : \mathit{String}, o : \mathit{String})$
local $l : \mathit{List}[\mathit{String}]$
init $\mathit{empty} \equiv \{((i, o), l) \mid l = [\,]\}$

ops $scons \equiv i = i' \wedge o = o' \wedge l' = Cons(i, l)$
$stl \equiv l \neq [\,] \wedge i = i' \wedge o = o' \wedge l' = tl(l)$
$shd \equiv l \neq [\,] \wedge i' = i \wedge o' = hd(l) \wedge l' = l$

The definitions resemble those of *ListStack*, but have different operation names in addition to the new type. The abstracted transformation rule is applied to this data structure and an array of strings is obtained. In detail, suppose that `StringList.sl` is the Isabelle constant defining *StringList* (corresponding to `ls` for *ListStack* above).

First, the variable `?a` in the transformation rule is instantiated with the *StringList* data structure `StringList.sl`. This instantiates the type of `globalstate` to the product of strings. Second, the variables for the list operations (such as `ltop`) are instantiated with the corresponding definitions from *StringList*. The result is a rule where the first proof obligations are exactly the definitions of the operations `stl`, `scons`, etc. These proof obligations are automatically discharged by resolution with the corresponding definitions `stl_def`, `scons_def`, etc. In the resulting theorem, the definition of the stack of strings as arrays can be retrieved from the applicability conditions of the rule:

```
?atop == {(((r, ra), ra, rb), (rc, rd), re, rf).
            rc = r & rd = the (rb (ra - 1))
                          & re = ra & rf = rb};
?apop == {((r, ra, rb), rc, rd, re). rc = r
                          & rd = ra - 1 & re = rb};
?apush == {(((r, ra), rb, rc), (rd, re), rf, rg).
            (rd, re) = (r, ra) & rf = rb + 1
                          & rg = rc(rb|->r)};
!!u. ?aops u == case u of push => ?apush
                          | pop => ?apop | top => ?atop;
?aempty == {(g, l). l = (0, Map.empty)};
?c == (?aempty, ?aops);
```

The above definitions completely define *StringArray*, because `?c` is the transformed specification. The transformation rule also provides a definition of the retrieve relation and a correctness proof for the transformation.

6. Related work

6.1 Transformational development systems

Several authors have studied implementations of transformation or window inference systems in theorem provers [Anderson and Basin 2000, Butler *et al.* 1997, Staples 1998, Carrington *et al.* 1998, Lüth and Wolff 2000, Grundy 1996, Hemer *et al.* 2001, Långbacka *et al.* 1995]. Some of these systems may be extended, e.g. PRT [Carrington *et al.* 1998] allows user addition of new transformation rules to the system by validity proofs, and TAS [Lüth and Wolff 2000] can be instantiated with arbitrary refinement relations.

A meta-logic approach to derive transformation rules is proposed by Anderson and Basin [2000], formulating and proving theorems directly in Isabelle's meta-logic. However their approach does not provide guidelines on how to construct transformation rules; in this respect they follow the invent and verify method. In contrast, the present paper shows how general meta-level theorems can be derived from theorems in object logics, providing a constructive strategy for deriving transformation rules by generalising examples of refinements that have already been proved.

6.2 *Abstraction and proof reuse*

Many approaches attempt to reuse parts of old proofs to solve new problems. In the context of formal program development, the KIV verification system reuses proof fragments to reprove old theorems after modifications to an initial program [Reif and Stenzel 1993]. 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 toward syntax-driven proof methods typical of program verification. In contrast, a semantic approach is taken in MAYA [Autexier *et al.* 2002], where specifications are represented by a development graph which distinguishes local and global axioms, theories, and links between them, in order to locate changes in the specifications. In a similar vein, Schairer and Hutter [2002] consider how proofs for theorems are affected by design changes in a specification. They propose a set of basic changes mirrored by proof transformations that restructure the proof tree, allowing open branches in the derived proofs. The approach allows addition and removal of datatype constructors, recursively leaving new open branches or removing branches to reflect the change. Whereas these approaches are concerned with reusing proof work in a modified specification, our aim is to derive widely applicable transformation rules from specific developments.

In a logical framework setting, Felty and Howe [1994] describe a generic approach to generalisation and reuse of tactic proofs. In their work, proof steps in λProlog consist of an inference rule and a substitution. 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. This allows reuse of steps from abortive proof attempts, e.g. wrong variable instantiations, which can to some extent be mimicked by considering different unifiers for our derived inference rules. However, their approach cannot handle type-specific proof rules.

In the type-theory based Coq prover, Magaud [2003] uses proof transformations to reuse proofs across types. This is done by first establishing correspondences between basic properties (definitions and lemmas) of the present and target type, after which properties of the present type can be systematically replaced in the proof by those of the target type before replaying the proof. A proof of a generalised theorem is not derived.

7. Conclusion and future work

The intuitive appeal of stepwise formal program development stands in sharp contrast to the cumbersome details of discharging the proof obligations connected to every individual development step. Such a development process needs to be supported by powerful tools. By abstracting single development steps to transformation rules, it is possible to reuse much of the proof work. This way it is possible to construct libraries of transformation rules for standard development steps in different problem domains. Using such transformation libraries, focus is kept on design rather than on proof, while correctness is automatically maintained for standard development steps.

This paper demonstrates an abstraction process that results in generalised transformation rules from specific examples of refinement. The rules are derived within the Isabelle theorem prover and the correctness of the rules is guaranteed by construction. The proposed abstraction process lifts the proof of a theorem into a correctness proof for a derived inference rule. The process is illustrated by deriving a polymorphic transformation rule for stacks from a traditional example of data refinement of stacks representations.

In future work, we want to integrate this abstraction method in the transformation system TAS, enabling the developer to improve the available library of transformation rules during the development process. Further, we are interested in case studies using the abstraction method to develop transformation rules in established and widely used specification formalisms. Finally, further improvements to the abstraction mechanism are of interest; for example, by identifying dependencies between different occurrences of a function symbol or a type, and by mechanisms for polytypic abstraction, making rules applicable for structures with a different number of constructors.

References

Anderson, Penny and Basin, David. 2000. Program Development Schemata as Derived Rules. *Journal of Symbolic Computation 30*, 1 (July), 5–36.

Autexier, Serge, Hutter, Dieter, Mossakowski, Till, and Schairer, Axel. 2002. The Development Graph Manager Maya. In *Proc. 9th Int. Conf. Algebraic Methodology and Software Technology (AMAST'02)*, Volume 2422 of *Lecture Notes in Computer Science*. Springer, 495– 501.

Back, Ralph-Johan, Grundy, Jim, and von Wright, Joakim. 1997. Structured Calculational Proof. *Formal Aspects of Computing 9*, 5–6, 469–483.

Back, Ralph-Johan and von Wright, Joakim. 1998. *Refinement Calculus: a Systematic Introduction*. Springer.

Bauer, Friedrich L., *et al.* 1985. *The Munich Project CIP. The Wide Spectrum Language CIP-L*. Lecture Notes in Computer Science. Springer, Berlin.

Berghofer, Stefan and Wenzel, Markus. 1999. Inductive Datatypes in HOL — Lessons Learned in Formal-Logic Engineering. In *13th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'99)*, Volume 1690 of *Lecture Notes in Computer Science*. Springer, 19–36.

Broy, Manfred. 1997. Compositional Refinement of Interactive Systems. *Journal of the ACM 44*, 6 (Nov.), 850–891.

Burstall, Rod M. and Darlington, John. 1977. A Transformational System for Developing Recursive Programs. *Journal of the ACM 24*, 1 (Jan.), 44–67.

Butler, Michael, Grundy, Jim, Långbacka, Thomas, Rukšėnas, Rimvydas, and von Wright, Joakim. 1997. The Refinement Calculator: Proof Support for Program Refinement. In *Proc. Formal Methods Pacific'97*. Springer, 40–61.

Carrington, David, Hayes, Ian, Nickson, Ray, Watson, Geoffrey, and Welsh, Jim. 1998. A Program Refinement Tool. *Formal Aspects of Computing 10*, 97–124.

Dahl, Ole-Johan, Myrhaug, Bjørn, and Nygaard, Kristen. 1968. (Simula 67) Common Base Language. Tech. Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway.

de Roever, Willem-Paul and Engelhardt, Kai. 1998. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, Volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY.

Derrick, John and Boiten, Eerke A. 2001. *Refinement in Z and Object-Z*. Formal Approaches to Computing and Information Technology (FACIT). Springer.

Felty, Amy and Howe, Douglas. 1994. Generalization and Reuse of Tactic Proofs. In *Fifth Int. Conf. on Logic Programming and Automated Reasoning (LPAR'94)*, Volume 822 of *Lecture Notes in Computer Science*. Springer, 1–15.

Grundy, Jim. 1996. Transformational Hierarchical Reasoning. *The Computer Journal 39*, 4 (May), 291–302.

Harper, Robert, Honsell, Furio, and Plotkin, Gordon. 1993. A Framework for Defining Logics. *Journal of the ACM 40*, 1 (Jan.), 143–184.

Hemer, David, Hayes, Ian, and Strooper, Paul. 2001. Refinement Calculus for Logic Programming in Isabelle/HOL. In *14th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'01)*, Volume 2152 of *Lecture Notes in Computer Science*. Springer, 249–264.

Hoare, C. A. R. 1972. Proofs of Correctness of Data Representations. *Acta Informatica 1*, 271–281.

Hoffmann, Berthold and Krieg-Brückner, Bernd. 1993. *PROSPECTRA: Program Development by Specification and Transformation*. Volume 690 of *Lecture Notes in Computer Science*. Springer.

Huet, Gérard P. and Lang, Bernard. 1978. Proving And Applying Program Transformations Expressed With Second-Order Patterns. *Acta Informatica 11*, 1, 31–55.

Johnsen, Einar Broch and Lüth, Christoph. 2003. Abstracting Theorems for Reuse. Submitted for publication.

Långbacka, Thomas, Rukšėnas, Rimvydas, and von Wright, Joakim. 1995. TkWinHOL: A Tool for Window Interference in HOL. In *8th Int. Workshop on Higher Order Logic Theorem Proving and its Applications*, Volume 971 of *Lecture Notes in Computer Science*. Springer, Aspen Grove, Utah, USA, 245–260.

Lüth, Christoph and Wolff, Burkhart. 2000. TAS — A Generic Window Inference System. In *13th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'00)*, Volume 1869 of *Lecture Notes in Computer Science*. Springer, 405–422.

Magaud, Nicolas. 2003. Changing Data Representation within the Coq System. In *16th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'2003)*, To appear in *Lecture Notes in Computer Science*. Springer.

Nipkow, Tobias, Paulson, Lawrence C., and Wenzel, Markus. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Volume 2283 of *Lecture Notes in Computer Science*. Springer.

Paulson, Lawrence C. 1990. Isabelle: The Next 700 Theorem Provers. In *Logic and Computer Science*, Odifreddi, Piergiorgio, Editor. Academic Press, 361–386.

Paulson, Lawrence C. 1997. Mechanizing Coinduction and Corecursion in Higher-order Logic. *Journal of Logic and Computation 7*, 2 (Apr.), 175–204.

Pfenning, Frank. 2001. Logical Frameworks. In *Handbook of Automated Reasoning*, Robinson, Alan and Voronkov, Andrei, Editors. Elsevier Publishers, 1063–1147.

Reif, Wolfgang, Schellhorn, Gerhard, Stenzel, Kurt, and Balser, Michael. 1998. Structured Specifications and Interactive Proofs with KIV. In *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques*, Bibel, Wolfgang and Schmidt, Peter H., Editors. Kluwer Academic Publishers.

Reif, Wolfgang and Stenzel, Kurt. 1993. Reuse of Proofs in Software Verification. In *Proc. of Foundations of Software Technology and Theoretical Computer Science*, Volume 761 of *Lecture Notes in Computer Science*. Springer, 284–293.

Roscoe, A. W. 1998. *The Theory and Practice of Concurrency*. Prentice Hall.

SANNELLA, DONALD. 2000. Algebraic Specification and Program Development by Stepwise Refinement. In *Proc. 9th Intl. Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99)*, Volume 1817 of *Lecture Notes in Computer Science*. Springer, 1–9.

SCHAIRER, AXEL AND HUTTER, DIETER. 2002. Proof Transformations for Evolutionary Formal Software Development. In *Proc. 9th Int. Conf. Algebraic Methodology and Software Technology (AMAST'02)*, Volume 2422 of *Lecture Notes in Computer Science*. Springer, 441–456.

SMITH, DOUGLAS R. 1985. The Design of Divide and Conquer Algorithms. *Science of Computer Programming 5*, 1 (Feb.), 37–58.

SMITH, DOUGLAS R. 1990. KIDS: a Semiautomatic Program Development System. *IEEE Transactions on Software Engineering 16*, 9 (Sept.), 1024–1043.

SMITH, DOUGLAS R. 1999. Mechanizing the Development of Software. In *Calculational System Design*, Broy, Manfred and Steinbrüggen, Rolf, Editors. Proc. of the Marktoberdorf Int. Summer School, NATO ASI Series. IOS Press, Amsterdam, 251–292.

STAPLES, MARK. 1998. *A Mechanised Theory of Refinement*. PhD thesis, University of Cambridge.

WOODCOCK, JIM C. P. AND DAVIES, JIM. 1996. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice Hall, New York, N.Y.

Appendix A. The derived transformation rule in Isabelle

This appendix presents the transformation rule derived in Section 5 of the paper. The rule is of the form $[\![A]\!] \Longrightarrow \mathbf{c} \sqsubseteq \mathbf{a}$ (where A represents the subgoals of the proof rule, cf. the modus ponens rule in Section 3.1). All the definitions and lemmas that were made explicit in the abstraction process can now be found as subgoals in the derived proof rule. In order to increase the readability of the rule, the formulas of A (the applicability conditions) have been arranged from top to bottom, first the definitions of `ListStack`, then the definitions of `ArrayStack`, and finally the definition of the retrieve relation, each group of definitions is followed by corresponding lemmas.

```
"[| ?lpop == {((r, ra), a, b). ra ~= [] & a = r & b = tl ra};
  ?lpush == {(((r, ra), rb), (rc, rd), re).
                 (rc, rd) = (r, ra) & re = r # rb};
  ?ltop == {(((i, out), l), (i', out'), l').
                  l ~= [] & i' = i & out' = hd l & l = l'};
  !!u. ?lops u == case u of push => ?lpush
                            | pop => ?lpop | top => ?ltop;
  ?lempty == {(g, l). l = []};
  ?a == (?lempty, ?lops);
  !!u ua ub. ?elems u ua = ub
               ==> ?elems (u - Suc 0) ua = tl ub;
  !!u ua ub uc. ?elems u ua = ub
                 ==> ?elems (Suc u) (ua(u|->uc)) = uc # ub;
  ?atop == {(((r, ra), ra, rb), (rc, rd), re, rf).
               rc = r & rd = the (rb (ra - 1))
                & re = ra & rf = rb};
  ?apop == {((r, ra, rb), rc, rd, re).
                            rc = r & rd = ra - 1 & re = rb};
  ?apush == {(((r, ra), rb, rc), (rd, re), rf, rg).
```

```
          (rd, re) = (r, ra) & rf = rb + 1 & rg = rc(rb|->r)};
    !!u. ?aops u == case u of push => ?apush
                              | pop => ?apop | top => ?atop;
    ?aempty == {(g, l). l = (0, empty)};
    ?c == (?aempty, ?aops);
    !!u ua ub. ((u, ua, ub) : ?r) = (u = ?elems ua ub);
    ?r == {(r, ra, rb). ra = length r &
                 (ALL rc. 0 <= rc & rc < ra
                      --> r ! rc = the (rb (ra - rc - 1)))} |]
==> ?c [= ?a" : thm
```

All function symbols have been abstracted in the transformation rule; the prefix '?' in the operation symbols is Isabelle's notation for meta-variables. The integer type of the example refinement has been abstracted and replaced by a type variable. For example, the operation `lpop` has been abstracted into

```
?lpop::(((?a_type * ?a_type) * ?a_type list)
           * (?a_type * ?a_type) * ?a_type list) set
```

where `?a_type` has replaced the integers, so `(?a_type * ?a_type)` represents the global space and `?a_type list` the local space before type instantiation. The type variable `?a_type` is instantiated along with the other meta-variables when the transformation rule is applied, using Isabelle's resolution techniques.