Theories, Implementations, and Transformations

Eric Hehner, Ioannis T. Kassios

Department of Computer Science, University of Toronto,
Toronto ON M5S 3G4 Canada
{hehner, ykass}@cs.utoronto.ca

Abstract. The purpose of this paper is to try to put theory presentation and structuring in the simplest possible logical setting in order to improve our understanding of it. We look at how theories can be combined, and compared for strength. We look at theory refinement and implementation, and what constitutes proof of correctness. Our examples come from both the functional style and imperative (state-changing) style of theory. Finally, we explore how one implementation can be transformed to another.

0 Introduction

A classic paper by Burstall and Goguen in 1977 [2] taught us to think about data types used in computer programs as logical theories, presented by axioms, whose properties can be explored by logical deduction. The following year, a paper by Guttag and Horning [4] developed the idea further, showing us the algebraic properties of data types presented as theories. Another important contribution came from Abrial [8] in the design of Z, and more recently B [1]. He brought to theory design all the structuring and scoping that programming languages provide, enabling us to build large theories by composing smaller ones. With the work of the Z and B community, and a change of terminology, theory design became an important part of software development.

The purpose of this paper is to try to put theory presentation and structuring in the simplest possible logical setting in order to improve our understanding of it. It is not the purpose of this paper to provide a notation or language for practical engineering use; for that task the Z and B community are the leaders.

1 Notation

Notation is not the point of this paper; as much as possible, we will use standard, or at least familiar, notations. The two booleans are $\top$ and $\bot$, and the boolean operators are $\neg \wedge \vee = \neq \Rightarrow \Leftarrow$. The same equality $=$ and unequality $\neq$ will be used with any type. We also use a large version $\mathrel{=\!\!=} \mathrel{\Longrightarrow} \mathrel{\Longleftarrow}$ of equality and implication that are identical to the small version except for their precedence; the only purpose is to save a clutter of parentheses. The empty bunch is *null*. The comma (,) is bunch union, which is commutative, idempotent, and associative. The colon (:) is bunch inclusion. For example,

$$2, 9 : 0, 2, 5, 9$$

is a boolean expression with value $\top$ because the left operand of colon is included in the right operand. We use the asymmetric notation $x,..y$ for the bunch of integers from and including x up to but excluding y. The empty list is $[nil]$, and the list $[2; 6; 4; 8]$ contains four items. The notation $[x;..y]$ is used for the list of integers from and including x up to but excluding y. Lists are indexed from 0. List formation distributes over bunch union, so if *nat* is the natural numbers, then $[nat]$ is the list whose one item is the bunch of natural numbers, or equally, the bunch of all lists whose one item is a natural number. A star denotes repetition of an item, so $[*nat]$ is all lists of natural numbers. We use $\#$ for list length. We use a standard lambda notation $\lambda x{:}\ D\cdot fx$ for functions, and juxtaposition for function application. We use $A{\to}B$ for the bunch of all functions with domain at least A and range at most B. Quantifiers $\forall\ \exists$ apply to functions, but for the sake of familiarity they replace the lambda.

Here are all the notations of the paper in a precedence table.

0.	$\top$ $\bot$ ( ) [ ] numbers names	(true, false, precedence, list brackets)
1.	juxtaposition	(function application) left-to-right
2.	$\#$ $*$ $\to$	(list length, item repetition, function space) right-to-left
3.	$+$ $-$ $^+$	(addition, subtraction, catenation) left-to-right
4.	; ;..	(sequencing of list items) associative
5.	, ,.. $\mid$	(bunch union, function selection) associative
6.	$=$ $\neq$ $<$ $>$ $\leq$ $\geq$:	(equality, unequality, order, inclusion) continuing
7.	$\neg$	(negation) right-to-left
8.	$\wedge$	(conjunction) associative

9.	∨	(disjunction) associative
10.	⇒ ⇐	(implication) continuing
11.	:=	(assignment)
12.	**if then else**	(if then else)
13.	;	(sequential composition) associative
14.	λ· ∀· ∃·	(function, quantifiers)
15.	= ⇒ ⇐	(equality, implication) continuing

To say that $=$ is continuing is to say that $a = b = c$ neither associates to the left nor associates to the right, but means $a = b \wedge b = c$. A mixture of continuing operators can be used; for example, $a \le b < c$ means $a \le b \wedge b < c$. For further details on notation and basic theories please consult [5] or [6].

2 Theories

Here is a little theory presented in a style similar to [2] and [4].

Theory0:	names:	*chain*, *start*, *link*, *isStart*
	signatures:	*start*: *chain*
		link: *chain*→*chain*
		isStart: *chain*→*bool*
	axioms:	*isStart start*
		∀*c*: *chain*· ¬ *isStart* (*link c*)

Theory0 introduces four new names into our vocabulary. The signatures section tells us something about the role these names will play in the theory. Then the axioms tell us what can be proven, what are the theorems, in this theory.

The first problem with this presentation of Theory0 is that names cannot be attached to theories. For example, this theory uses the name *bool*, and many others do too, and each of them is telling us something about *bool*. And when we build large theories by composing smaller ones, no particular theory in the composition can claim a name as its own. And it isn't just names that get introduced by theories; symbols like ≤, or in our example ∀ and ¬, and even =, are used in many theories, and each of them is telling us something more about the use of those symbols. Names and symbols are defined by their use in all theories where they appear; and we can always add more theories to the collection. As part of a library of

theories, we need a linked, browsable dictionary of names and symbols, telling us which theories use them. This dictionary should be generated automatically from the library of theories, so that it is always up-to-date. The first change to theory presentation is to remove the list of names.

The next change to theory presentation is to consider a signature to be a kind of boolean expression. One of the uses of Bunch Theory is as a fine-grained type theory. The boolean expression

$$5:\ 0, 3, 5, 8$$

has value $\top$ and says, "5 is included among 0, 3, 5, 8". But we can also read it as "5 has type 0, 3, 5, 8". Defining *nat* as the bunch of all natural numbers, the boolean expression $5: nat$ has value $\top$. And so $x: nat$ can be given as an axiom about x. So too $x, y: nat$ can be an axiom, just as $3, 5:\ 0, 3, 5, 8$ has value $\top$. The expression $A \to B$ consists of all functions with domain at least A and range at most B. For example,

$$(\lambda n: nat \cdot n+1) :\ nat \to nat$$

has value $\top$. And so $f: nat \to nat$ can be an axiom about f. By "currying", $A \to B \to C$ consists of two-variable functions, and so on.

The final change to theory presentation is just to write all the axioms as one big axiom by taking their conjunction. Now a theory consists of one single axiom, so there is now no difference between a theory and an axiom. Theory0 can be written as follows.

$$\begin{array}{lll} Theory0 & = & start: chain \\ & & \wedge\ link: chain \to chain \\ & & \wedge\ isStart: chain \to bool \\ & & \wedge\ isStart\ start \\ & & \wedge\ \forall c: chain \cdot \neg\ isStart\ (link\ c) \end{array}$$

3 Composition

The original paper by Burstall and Goguen [2] presents four operations on theories: combination, enrichment, induction, and derivation. To illustrate theory

combination, here is a second theory.

$$\begin{array}{lll} \mathit{Theory1} & = & \mathit{start}\colon \mathit{chain} \\ & \wedge & \mathit{link}\colon \mathit{chain} \to \mathit{chain} \\ & \wedge & (\forall c\colon \mathit{chain}\cdot\ \mathit{start} \neq \mathit{link}\ c) \\ & \wedge & (\forall c, d\colon \mathit{chain}\cdot\ (c{=}d) = (\mathit{link}\ c = \mathit{link}\ d)) \end{array}$$

Theory0 and *Theory1* have much in common, but also some differences; there are theorems in each that are not theorems in the other. With our form of theory presentation, we can combine the two theories with ordinary boolean conjunction.

$$\mathit{Theory2} \ = \ \mathit{Theory0} \wedge \mathit{Theory1}$$

Burstall and Goguen's next theory operation, enrichment, is also just conjunction, but with further axioms rather than with a named theory. Here is an example.

$$\begin{array}{lll} \mathit{Theory3} & = & \mathit{Theory2} \\ & \wedge & \forall c\colon \mathit{chain}\cdot\ \mathit{start} \leq c < \mathit{link}\ c \end{array}$$

The next of Burstall and Goguen's theory operations adds a structural induction scheme over the generators of the new data type. For us, it is again just conjunction of another axiom.

$$\begin{array}{lll} \mathit{Theory4} & = & \mathit{Theory3} \\ & \wedge & \forall P\colon (\mathit{chain} \to \mathit{bool})\cdot \\ & & \quad P\ \mathit{start} \wedge (\forall c\colon \mathit{chain}\cdot\ P\ c \Rightarrow P\ (\mathit{link}\ c)) \\ & \Rightarrow & \quad \forall c\colon \mathit{chain}\cdot\ P\ c \end{array}$$

That is the familiar form of induction; a neater, equivalent form is as follows.

$$\begin{array}{lll} \mathit{Theory4} & = & \mathit{Theory3} \\ & \wedge & \forall C\cdot\ \mathit{start}, \mathit{link}\ C\colon C \ \Rightarrow\ \mathit{chain}\colon C \end{array}$$

To briefly explain this axiom, most operators and functions distribute over bunch union. For example,

$$(2, 5, 9) + 1 \ = \ (3, 6, 10)$$

So *link C* consists of all the results of applying *link* to things in *C*. The axiom says that if *start* and all the links of things in *C* are included in *C*, then *chain* is included in *C*. The antecedent can be rewritten as

$$start: C \quad \wedge \quad link: C \to C$$

and, regarding *C* as the unknown, *chain* is one solution. The axiom therefore says that *chain* is the smallest solution.

Burstall and Goguen's final operation on theories, derivation, allows part of a theory to be hidden from the theory users. For us, that's existential quantification.

$$Theory5 \quad = \quad \exists start: chain \cdot Theory4$$

Theory5 has all the same theorems as *Theory4* minus those that mention *start*. If we want to keep all the theorems of *Theory4* but rename *start* as *new*, define

$$Theory6 \quad = \quad \exists start: chain \cdot start = new \wedge Theory4$$

We can combine theories with other boolean operators too, such as disjunction and implication. For example,

$$Theory7 \quad = \quad (\forall c: chain \cdot new \le c) \Rightarrow Theory6$$

This makes *Theory7* such that if we had the axiom $\forall c: chain \cdot new \le c$ then we would have *Theory6*. In a vague sense, *Theory7* is *Theory6* without $\forall c: chain \cdot new \le c$. To be precise, if we take *Theory7* and add the axiom $\forall c: chain \cdot new \le c$, we get back *Theory6*.

$$Theory7 \wedge \forall c: chain \cdot new \le c \quad = \quad Theory6 \wedge \forall c: chain \cdot new \le c$$

New theories are not always built by additions to old theories; sometimes they are built by deletions. One of the problems with object-orientation is that, although subclassing allows us to add attributes, there is no way to delete attributes and make a superclass, nor to make an interclass between two existing classes.

These examples illustrate that our theory presentation is both a simplification and a generalization of the early work. By reducing theories to boolean expressions we understand them in the simplest possible way, and we allow all combinations that make logical sense.

4 Refinement and Implementation

A theory can serve as a specification of a data type, and of computation in general. Specifications can be refined, usually by resolving nondeterminism. Specification A refines specification B if all computer behavior satisfying A also satisfies B. If theories are expressed as single boolean expressions,

theory A refines theory B	means	$A \Rightarrow B$
theory B is refined by theory A	means	$B \Leftarrow A$

Refinement is just implication. So far, we have

$$
\begin{aligned}
&\mathit{Theory6} \Rightarrow \mathit{Theory7}\\
&\mathit{Theory4} \Rightarrow \mathit{Theory5}\\
&\mathit{Theory4} \Rightarrow \mathit{Theory3}\\
&\mathit{Theory3} \Rightarrow \mathit{Theory2}\\
&\mathit{Theory2} \Rightarrow \mathit{Theory1}\\
&\mathit{Theory2} \Rightarrow \mathit{Theory0}
\end{aligned}
$$

When we define a theory, and especially when we combine theories, there is always the danger of inconsistency. The only way to prove the consistency of a theory is to implement it. As software engineers, our goal is to design useful theories (they must be consistent to be useful), and to implement them. A theory is said to be implemented when all names and symbols appearing in it have been implemented. A name or symbol is implemented by defining it in terms of other names and symbols that are implemented. Ultimately, the computing machinery provides the ground theory on top of which all other theories are implemented. (To logicians, an implementation is known as a "model", and the ultimate machinery is usually taken to be set theory, although they might claim that the model is the sets themselves and not set theory.)

An implementation can be expressed in exactly the same form as a theory: a boolean expression. Here is an example implementation of *Theory4*, assuming that *nat* is an implemented data type, and that functions are implemented.

$$
\begin{aligned}
\mathit{Imp} \quad = \quad & \mathit{chain} = \mathit{nat}\\
\wedge \quad & \mathit{start} = 0\\
\wedge \quad & \mathit{isStart} = (\lambda c\colon \mathit{nat}\cdot\ c{=}0)\\
\wedge \quad & \mathit{link} = (\lambda c\colon \mathit{nat}\cdot\ c{+}1)
\end{aligned}
$$

An implementation is also a theory, but of a particular form. It is a conjunction of equations, and each equation has a left side consisting of one of the names needing an implementation, and a right side employing only names and symbols that are already implemented.

The benefit in expressing an implementation in the same form as a theory is that the proof of correctness of the implementation is now just a boolean implication. We prove that *Imp* correctly implements *Theory4* by proving

$$Imp \Rightarrow Theory4$$

Implementation is just refinement by an implemented theory. By the transitivity of implication we have immediately that *Imp* also implements *Theory5* , *Theory3* , *Theory2* , *Theory1* , and *Theory0* .

5 Functional Stack

From a typical mathematician's viewpoint, a stronger theory is a better theory because it allows us to prove more. But the theory must not be so strong as to be inconsistent, for then we can prove everything trivially. The game is to add axioms, approaching the brink of inconsistency as closely as possible without falling over. For example, here a strong but consistent theory of stacks.

$$\begin{array}{lll} Stack0 = \lambda X\cdot & & empty\colon stack \\ & \wedge & push\colon stack \to X \to stack \\ & \wedge & pop\colon stack \to stack \\ & \wedge & top\colon stack \to X \\ & \wedge & (\forall S\cdot\ empty, push\ S\ X\colon S \Rightarrow stack\colon S) \\ & \wedge & (\forall s\colon stack\cdot\ \forall x\colon X\cdot\ push\ s\ x \neq empty) \\ & \wedge & (\forall s, t\colon stack\cdot\ \forall x, y\colon X\cdot \\ & & \quad push\ s\ x = push\ t\ y = s{=}t \wedge x{=}y) \\ & \wedge & (\forall s\colon stack\cdot\ \forall x\colon X\cdot\ pop\ (push\ s\ x) = s) \\ & \wedge & (\forall s\colon stack\cdot\ \forall x\colon X\cdot\ top\ (push\ s\ x) = x) \end{array}$$

And here is an implementation, assuming lists, functions, and integers are already implemented.

$$
\begin{aligned}
\mathit{Stack1} \triangleq \quad & \mathit{stack} = [{*}\mathit{int}] \\
\wedge\ & \mathit{empty} = [\mathit{nil}] \\
\wedge\ & \mathit{push} = (\lambda s{:}\ \mathit{stack}\cdot\ \lambda x{:}\ \mathit{int}\cdot\ s{+}[x]) \\
\wedge\ & \mathit{pop} = (\lambda s{:}\ \mathit{stack}\cdot\ \textbf{if}\ s{=}\mathit{empty}\ \textbf{then}\ \mathit{empty}\ \textbf{else}\ s\ [0;..\#s{-}1]) \\
\wedge\ & \mathit{top} = (\lambda s{:}\ \mathit{stack}\cdot\ \textbf{if}\ s{=}\mathit{empty}\ \textbf{then}\ 0\ \textbf{else}\ s\ (\#s{-}1))
\end{aligned}
$$

where $[{*}\mathit{int}]$ is all lists of integers, $[\mathit{nil}]$ is the empty list, $+$ is catenation, $\#$ is length, and $s\ [0;..\#s{-}1]$ is list s up to but not including its last item. To prove that *Stack1* is an implementation of *Stack0* we must prove

$$\mathit{Stack1} \Rightarrow \mathit{Stack0}\ \mathit{int}$$

but we won't spend the space here.

The only way to prove the consistency of a theory is to implement it. The only way to prove the incompleteness of a theory is to implement it twice such that some boolean expression is a theorem of one implementation, and its negation is a theorem of the other. In our example,

$$
\begin{aligned}
&\mathit{pop}\ \mathit{empty} = \mathit{empty} \\
&\mathit{top}\ \mathit{empty} = 0
\end{aligned}
$$

are theorems of *Stack1* . But here is another implementation of *Stack0 int* :

$$
\begin{aligned}
\mathit{Stack2} \quad = \quad & \mathit{stack} = [{*}\mathit{int}] \\
\wedge\ & \mathit{empty} = [\mathit{nil}] \\
\wedge\ & \mathit{push} = (\lambda s{:}\ \mathit{stack}\cdot\ \lambda x{:}\ \mathit{int}\cdot\ s{+}[x]) \\
\wedge\ & \mathit{pop} = (\lambda s{:}\ \mathit{stack}\cdot\ \textbf{if}\ s{=}\mathit{empty}\ \textbf{then}\ \mathit{push}\ \mathit{empty}\ 0 \\
& \qquad\qquad\qquad\qquad \textbf{else}\ s\ [0;..\#s{-}1]) \\
\wedge\ & \mathit{top} = (\lambda s{:}\ \mathit{stack}\cdot\ \textbf{if}\ s{=}\mathit{empty}\ \textbf{then}\ 1\ \textbf{else}\ s\ (\#s{-}1))
\end{aligned}
$$

in which their negations are theorems. So *Stack0 int* is incomplete. That means we can find a stronger theory of stacks by saying what *pop empty* and *top empty* are. But do we want a stronger theory? What is the purpose of this theory?

In *Stack0* , we have *empty*: *stack* and *pop*: *stack*→*stack* ; from them we can prove *pop empty*: *stack* . In other words, popping the empty stack gives a stack, though we do not know which one. An implementer is obliged to give a stack for *pop empty* , though it does not matter which one. If we never want to pop an empty

stack, then the theory is too strong. We should weaken the conjunct $pop: stack \to stack$ and remove the implementer's obligation to provide something that is not wanted. The weaker conjunct

$$\forall s: stack \cdot s \neq empty \Rightarrow pop\ s: stack$$

says that popping a nonempty stack yields a stack, but it is implied by the remaining conjuncts and is unnecessary. Similarly from $empty: stack$ and $top: stack \to X$ we can prove $top\ empty: X$; deleting $top: stack \to X$ removes an implementer's obligation to provide an unwanted result for $top\ empty$.

We may decide that we have no need to prove anything about all stacks, and can do without induction $\forall S \cdot empty, push\ S\ X: S \Rightarrow stack: S$. After a little thought, we may realize that we never need an empty stack, nor to test if a stack is empty. We can always work on top of a given (possibly non-empty) stack, and in most uses we are required to do so, leaving the stack as we found it. We can delete $empty: stack$ and all mention of $empty$. We must replace it with the weaker $stack \neq null$ so that we can still declare variables of type $stack$. If we do want to test whether a stack is empty, we can begin by pushing some special value, one that will not be pushed again, onto the stack; the empty test is then a test whether the top is the special value.

For most purposes, it is sufficient to be able to push items onto a stack, pop items off, and look at the top item. The theory we need is considerably simpler than the one presented previously.

$$\begin{aligned} Stack3 = \lambda X \cdot \quad & stack \neq null \\ \wedge \quad & (\forall s: stack \cdot \forall x: X \cdot push\ s\ x: stack) \\ \wedge \quad & (\forall s: stack \cdot \forall x: X \cdot pop\ (push\ s\ x) = s) \\ \wedge \quad & (\forall s: stack \cdot \forall x: X \cdot top\ (push\ s\ x) = x) \end{aligned}$$

For the purpose of studying stacks, as a mathematical activity, we want a strong theory so that we can prove as much as possible. As an engineering activity, theory design is the art of excluding all unwanted implementations while allowing all the others. It is counter-productive to design a stronger theory than necessary; it makes implementation harder, and it makes theory extension harder.

6 Imperative Stack

It is an accident of history that the usual stack specification is functional in style, while the usual stack implementation is imperative. Functions were familiar mathematics, suitable for formal specification, at a time when imperative programs were still understood only as commands for the operation of a computer. We now have a mathematical understanding of imperative, state-changing programs. We can equally well have specifications that are both mathematical and imperative.

In the simplest version of imperative stack theory, *push* is a procedure with parameter of type X, *pop* is a program, and *top* is an expression of type X. In this theory, *push* 3 is a program (assuming 3: X); it changes the state. Following this program, before any other pushes and pops, *print top* will print 3. Here is the theory.

$$\begin{array}{lll} \mathit{Stack4} & = & \forall x{:}\, X\cdot \quad (\mathit{top}'{=}x \Leftarrow \mathit{push}\ x) \\ & & \wedge \quad (\mathit{ok} \Leftarrow \mathit{push}\ x;\ \mathit{pop}) \end{array}$$

The first conjunct says that following a push, the top tem is the item pushed. In the second conjunct, *ok* (sometimes called *skip*) is a program (which is a specification, which is a boolean expression) that says that all final values of variables equal the corresponding initial values (the identity relation on states). So the second conjunct says that a pop undoes a push. In fact, it says that any natural number of pushes are undone by the same number of pops.

$$\begin{array}{lll} & \mathit{ok} & \text{use } \mathit{ok} \Leftarrow \mathit{push}\ x;\ \mathit{pop} \\ \Leftarrow & \mathit{push}\ x;\ \mathit{pop} & \mathit{ok} \text{ is identity for sequential composition} \\ = & \mathit{push}\ x;\ \mathit{ok};\ \mathit{pop} & \text{Reuse } \mathit{ok} \Leftarrow \mathit{push}\ x; \mathit{pop} \text{ and ; is monotonic} \\ \Leftarrow & \mathit{push}\ x;\ \mathit{push}\ y;\ \mathit{pop};\ \mathit{pop} & \end{array}$$

We can prove things like

$$\mathit{top}'{=}x \Leftarrow \mathit{push}\ x;\ \mathit{push}\ y;\ \mathit{push}\ z;\ \mathit{pop};\ \mathit{pop}$$

which say that when we push something onto the stack, we find it there later at the appropriate time. That is all we really want from a stack.

If we need only one stack, we obtain an economy of expression and of execution by leaving it implicit, as in *Stack4*. There is no need to say which stack to push

onto if there is only one. (If we need more than one stack, we can add an extra parameter to each operation.)

In imperative theories, the state is divided into two kinds of variables: the user's variables and the implementer's variables. A user of the theory enjoys full access to the user's variables, but cannot directly access (see or change) the implementer's variables. A user gets access to the implementer's variables only through the theory. On the other side, an implementer of the theory enjoys full access to the implementer's variables, but cannot directly access (see or change) the user's variables. An implementer gets access to the user's variables only through the theory.

To implement *Theory4*, we introduce an implementer's variable s: $[*X]$ and now we define

$$\begin{array}{lll} \mathit{Stack5} & = & (\mathit{push} = \lambda x{:}\, X \cdot s := s{+}[x]) \\ & \wedge & (\mathit{pop} = s := s\ [0;..\#s{-}1]) \\ & \wedge & (\mathit{top} = s\ (\#s{-}1)) \end{array}$$

The proof that *Stack5* implements *Stack4*, as always, is just an implication.

$$\mathit{Stack5} \Rightarrow \mathit{Stack4}$$

By implementing *Stack4* we prove that it is consistent. But it is incomplete. Incompleteness is a freedom for the implementer, who can trade economy against robustness. If we care how this trade will be made, we should strengthen the theory. For example, we could add

$$\begin{array}{lll} \mathit{Stack6} & = & \mathit{Stack4} \\ & \wedge & (\mathit{print}\ \text{"error"} \Leftarrow \mathit{mkempty};\ \mathit{pop}) \end{array}$$

A slightly fancier imperative stack theory tells us about *mkempty* (a program to make the stack empty) and *isempty* (a boolean to say whether the stack is empty). Letting x: X, the theory is

$$\begin{array}{lll} \mathit{Stack7} & = & \mathit{Stack4} \\ & \wedge & (\forall x{:}\, X \cdot \neg\, \mathit{isempty}' \Leftarrow \mathit{push}\ x) \\ & \wedge & (\mathit{isempty}' \Leftarrow \mathit{mkempty}) \end{array}$$

The imperative stack theory we presented first, *Stack4* , can be weakened and still retain its stack character. We must keep

$$top'=x \Leftarrow push\ x$$

but we do not need the composition $push\ x; pop$ to leave all variables unchanged. We do require that any natural number of pushes followed by the same number of pops gives back the original top. The theory is

$$\begin{aligned} Stack8 \;=\; \exists balance\cdot \quad & (top'=x \Leftarrow push\ x) \\ \wedge\ & (top'=top \Leftarrow balance) \\ \wedge\ & (balance = ok \vee \exists x\cdot (push\ x;\ balance;\ pop)) \end{aligned}$$

This weaker theory allows an implementation in which popping does not restore the implementer's variable s to its pre-pushed value, but instead marks the last item as "garbage".

A weak theory can be extended in ways that are excluded by a strong theory. For example, we can add the names *count* (of type *nat*) and *start* (a program), as follows:

$$\begin{aligned} Stack9 \;=\; & Stack8 \\ \wedge\ & (count' = 0 \Leftarrow start) \\ \wedge\ & (\forall x: X\cdot\ count' = count+1 \Leftarrow push\ x) \\ \wedge\ & (count' = count+1 \Leftarrow pop) \end{aligned}$$

so that *count* counts the number of pushes and pops. From a software engineering point of view, the weakest theory is best.

7 Functional Tree

Here is a strong theory that is good for mathematicians who want to study trees.

$$\begin{aligned} Tree0 \;=\; \lambda X\cdot \quad & emptree: tree \\ \wedge\ & graft: tree \to X \to tree \to tree \\ \wedge\ & (\forall T\cdot\ emptree, graft\ T\ X\ T: T \Rightarrow tree: T) \\ \wedge\ & (\forall t, u: tree\cdot\ \forall x: X\cdot\ graft\ t\ x\ u \neq emptree) \end{aligned}$$

$$\begin{array}{ll} & (\forall t, u, v, w: tree \cdot \forall x, y: X \cdot \\ & \quad graft\ t\ x\ u = graft\ v\ y\ w \ = \ t{=}v \wedge x{=}y \wedge u{=}w) \\ \wedge & (\forall t, u: tree \cdot \forall x: X \cdot left\ (graft\ t\ x\ u) = t) \\ \wedge & (\forall t, u: tree \cdot \forall x: X \cdot root\ (graft\ t\ x\ u) = x) \\ \wedge & (\forall t, u: tree \cdot \forall x: X \cdot right\ (graft\ t\ x\ u) = u) \end{array}$$

For programming purposes, a simpler, weaker theory is sufficient. As with stacks, we don't really need to be given an empty tree. As long as we are given some tree, we can build a tree with a distinguished root that serves the same purpose. And we probably don't need tree induction.

$$\begin{array}{lll} Tree1 \ = \ \lambda X \cdot & & tree \neq null \\ & \wedge & (\forall t, u: tree \cdot \forall x: X \cdot graft\ t\ x\ u: tree) \\ & \wedge & (\forall t, u: tree \cdot \forall x: X \cdot left\ (graft\ t\ x\ u) = t) \\ & \wedge & (\forall t, u: tree \cdot \forall x: X \cdot root\ (graft\ t\ x\ u) = x) \\ & \wedge & (\forall t, u: tree \cdot \forall x: X \cdot right\ (graft\ t\ x\ u) = u) \end{array}$$

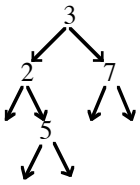
If lists and recursive data definition are implemented, then we can implement a tree of integers by the following theory.

$$\begin{array}{lll} Tree2 \ = & & tree \ = \ emptree, graft\ tree\ int\ tree \\ & \wedge & emptree \ = \ [nil] \\ & \wedge & (graft \ = \ \lambda t: tree \cdot \lambda x: int \cdot \lambda u: tree \cdot \ [t; x; u]) \\ & \wedge & (left \ = \ \lambda t: tree \cdot \ t\ 0) \\ & \wedge & (right \ = \ \lambda t: tree \cdot \ t\ 2) \\ & \wedge & (root \ = \ \lambda t: tree \cdot \ t\ 1) \end{array}$$

Here is another implementation.

$$\begin{array}{lll} Tree3 \ = & & tree \ = \ emptree, graft\ tree\ int\ tree \\ & \wedge & emptree \ = \ 0 \\ & \wedge & (graft \ = \ \lambda t: tree \cdot \lambda x: int \cdot \lambda u: tree \cdot \\ & & \qquad \text{"left"} \rightarrow t \mid \text{"root"} \rightarrow x \mid \text{"right"} \rightarrow u) \\ & \wedge & (left \ = \ \lambda t: tree \cdot \ t\ \text{"left"}) \\ & \wedge & (right \ = \ \lambda t: tree \cdot \ t\ \text{"right"}) \\ & \wedge & (root \ = \ \lambda t: tree \cdot \ t\ \text{"root"}) \end{array}$$

According to *Tree2* , the tree



is

[[[*nil*]; 2; [[*nil*]; 5; [*nil*]]]; 3; [[*nil*]; 7; [*nil*]]]

and according to *Tree3* it is

```
"left" →     ("left" → 0
                | "root" → 2
                | "right" →    ("left" → 0
                                | "root" → 5
                                | "right" → 0 ) )
| "root" → 3
| "right" →   ("left" → 0
                | "root" → 7
                | "right" → 0 )
```

Both *Tree2* and *Tree3* implement *Tree0* , and therefore also *Tree1* .

Tree2 ∨ *Tree3* ⇒ *Tree0 int* ⇒ *Tree1 int*

8 Imperative Tree

Imagine a tree that is infinite in all directions; there are no leaves and no root. You are standing at one node in the tree facing one of the three directions *up* (towards the parent of this node), *left* (towards the left child of this node), or *right* (towards the right child of this node). Variable *node* (of type *X*) tells the value of the item where you are, and it can be assigned a new value. Variable *aim* tells what direction you are facing, and it can be assigned a new direction. Program *go* moves you to the next node in the direction you are facing, and turns you facing back the way you

came. For example, we might begin with

aim:= *up*; *go*

and then look at *aim* to see where we came from. For later use, we might then assign

node:= 3

The theory uses an auxiliary definition: *work* means "Do anything, wander around changing the values of nodes if you like, but do not *go* from this node (your location at the start of *work*) in this direction (the value of variable *aim* at the start of *work*). End where you started, facing the way you were facing at the start.".

$$
\begin{array}{lll}
\mathit{Tree4} = \exists \mathit{work}\cdot & & \\
\quad & ((\mathit{aim}{=}\mathit{up}) = (\mathit{aim}'{\neq}\mathit{up}) \Leftarrow \mathit{go}) & \\
\quad \wedge & (\mathit{node}'{=}\mathit{node} \wedge \mathit{aim}'{=}\mathit{aim} \Leftarrow \mathit{go};\ \mathit{work};\ \mathit{go}) & \\
\quad \wedge & (\mathit{work} = \mathit{ok} & \\
\quad & \quad \vee\ (\exists x\cdot\ \mathit{node}{:=}\ x) & \\
\quad & \quad \vee\ (\exists a, b{:}\ \mathit{up}, \mathit{left}, \mathit{right}\cdot\ a{=}\mathit{aim}{\neq}b & \\
\quad & \quad\quad \wedge\ (\mathit{aim}{:=}b;\ \mathit{go};\ \mathit{work};\ \mathit{go};\ \mathit{aim}{:=}\ a)) & \\
\quad & \quad \vee\ (\mathit{work};\ \mathit{work})) &
\end{array}
$$

9 Transformation

A program is a specification of computer behavior. Sometimes (but not always) a program is the clearest kind of specification. Sometimes it is the easiest kind of specification to write. If we write a specification as a program, there is no work to implement it.

Even though a specification may already be a program, we can, if we like, implement it differently. An imperative theory is presented in terms of user's variables and implementer's variables; the former provide the user's interface to the theory; the latter may be for implementation purposes or they may just be for explanatory purposes. Perhaps the implementer's variables were chosen to make the specification as clear as possible, but other implementer's variables might be more storage-efficient, or provide faster access on average. Since a theory user has no access to the implementer's variables except through the theory, an implementer is free to change them in any way that provides the same theory to the user.

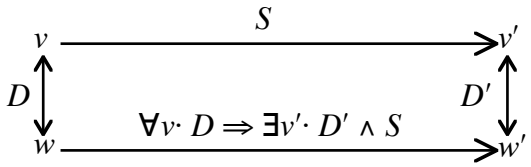
Let the user's variables be u, and let the implementer's variables be v (u and v represent any number of variables). Now suppose we want to replace the

implementer's variables by new implementer's variables w. We accomplish this transformation by means of a transformer, which is a boolean expression D relating v and w such that $\forall w \cdot \exists v \cdot D$. Let D' be the same as D but with primes on all the variables. Then each specification S in the theory is transformed to

$$\forall v \cdot D \Rightarrow \exists v' \cdot D' \wedge S$$

Specification S is in variables u and v, and the transformed specification is in variables u and w.

Transformation is invisible to the user. The user imagines that the implementer's variables are initially in state v, and then, according to specification S, they are finally in state v'. Actually, the implementer's variables will initially be in state w related to v by D; the user will be able to suppose they are in a state v because $\forall w \cdot \exists v \cdot D$. The implementer's variables will change state from w to w' according to the transformed specification $\forall v \cdot D \Rightarrow \exists v' \cdot D' \wedge S$. This says that whatever related initial state v the user was imagining, there is a related final state v' for the user to imagine as the result of S, and so the fiction is maintained. Here is a picture of it.



Implementability of S (in its variables v and v') becomes (via the transformer D and D') the new specification (in the new variables w and w'). This transformation is one form of data refinement.

10 Limited Queue

We illustrate theory transformation with the example of an imperative queue of limited size. Here's the theory we start with.

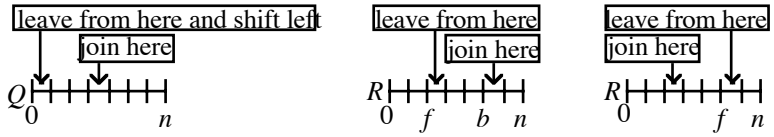
$$
\begin{array}{l}
Queue0 \;=\; \\
\quad \forall x\colon X\cdot \quad (mkemptyq \Rightarrow isemptyq') \\
\qquad \wedge\ (isemptyq \wedge \neg isfullq \wedge join\ x \;\Rightarrow\; front'{=}x \wedge \neg isemptyq') \\
\qquad \wedge\ (\neg isemptyq \wedge leave \;\Rightarrow\; \neg isfullq') \\
\qquad \wedge\ (\neg isemptyq \wedge \neg isfullq \wedge join\ x \;\Rightarrow\; front'{=}front \wedge \neg isemptyq') \\
\qquad \wedge\ (isemptyq \wedge \neg isfullq \;\Rightarrow\; (join\ x;\ leave \;=\; mkemptyq)) \\
\qquad \wedge\ (\neg isemptyq \wedge \neg isfullq \;\Rightarrow\; (join\ x;\ leave \;=\; leave;\ join\ x))
\end{array}
$$

Let the limit be positive natural n, and let $Q\colon [n{*}X]$ and $p\colon nat$ be implementer's variables. Then here is a theory to implement $Queue0$.

$$
\begin{array}{lll}
Queue1 \;=\; & (mkemptyq \;=\; p{:=}\ 0) \\
\wedge & (isemptyq \;=\; p{=}0) \\
\wedge & (isfullq \;=\; p{=}n) \\
\wedge & (join \;=\; \lambda x\colon X\cdot\ Qp{:=}\ x;\ p{:=}\ p{+}1) \\
\wedge & (leave \;=\; \textbf{for}\ i{:=}\ 1;..p\ \textbf{do}\ Q(i{-}1){:=}\ Qi;\ p{:=}\ p{-}1) \\
\wedge & (front \;=\; Q0)
\end{array}
$$

A user of $Queue1$ would be well advised to precede any use of $join$ with the test $\neg isfullq$, and any use of $leave$ or $front$ with the test $\neg isemptyq$, but that's not our business at the moment. A new item joins the back of the queue at position p taking constant time to do so. The front item is always found instantly at position 0. Unfortunately, removing the front item from the queue takes time $p{-}1$ to shift all remaining items down one index.

We want to transform the queue so that all operations are instant. Variables Q and p will be replaced by $R\colon [n{*}X]$ and $f, b\colon 0,..n$ with f indicating the current front of the queue and b its back.



The idea is that b and f move cyclically around the list; when f is to the left of b the queue items are between them; when b is to the left of f the queue items are in the outside portions.

Here is the transformer D.

$$\begin{array}{ll} & 0 \le p = b-f < n \ \wedge \ Q[0;..p] = R[f;..b] \\ \vee & 0 < p = n-f+b \le n \ \wedge \ Q[0;..p] = R[(f;..n);\ (0;..b)] \end{array}$$

One great thing about theory transformation is that once we have stated the transformer, which is the relation between the old and new variables, there is no further invention required; the operations of the theory are transformed for us. First we transform *mkemptyq* .

$$\begin{array}{ll} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge mkemptyq \\ = & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge p'=0 \wedge Q'=Q \quad \text{by several omitted steps} \\ = & f'=b' \\ \Leftarrow & f:= 0;\ b:= 0 \end{array}$$

The other great thing about theory transformation is that it never transforms incorrectly, even if we have an incorrect transformer! Next we transform $u:= isemptyq$.

$$\begin{array}{ll} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge (u:= isemptyq) \\ = & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge u'=(p=0) \wedge p'=p \wedge Q'=Q \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by several omitted steps} \\ = & \quad f<b \ \wedge \ f'<b' \ \wedge \ b-f = b'-f' \ \wedge \ R[f;..b] = R'[f';..b'] \ \wedge \ \neg u' \\ & \vee \quad f<b \ \wedge \ f'>b' \ \wedge \ b-f = n+b'-f' \\ & \quad \wedge \ R[f;..b] = R'[(f';..n);\ (0;..b')] \ \wedge \ \neg u' \\ & \vee \quad f>b \ \wedge \ f'<b' \ \wedge \ n+b-f = b'-f' \\ & \quad \wedge \ R[(f;..n);\ (0;..b)] = R'[f';..b'] \ \wedge \ \neg u' \\ & \vee \quad f>b \ \wedge \ f'>b' \ \wedge \ b-f = b'-f' \\ & \quad \wedge \ R[(f;..n);\ (0;..b)]=R'[(f';..n);\ (0;..b')] \ \wedge \ \neg u' \end{array}$$

Initially R might be in the "inside" or "outside" configuration, and finally R' might be either way, so that gives us four disjuncts. Very suspiciously, we have $\neg u'$ in every case. That's because $f=b$ is missing! So the transformed operation is unimplementable. That's the transformer's way of telling us that the new variables do not hold enough information to answer whether the queue is empty. The problem occurs when $f=b$ because that could be either an empty queue or a full queue. A solution is to add a new variable $m: bool$ to say whether we have the "inside" mode or "outside" mode. We revise the transformer D as follows:

$$\begin{array}{ll} & m \ \wedge \ 0 \le p = b-f < n \ \wedge \ Q[0;..p] = R[f;..b] \\ \vee & \neg m \ \wedge \ 0 < p = n-f+b \le n \ \wedge \ Q[0;..p] = R[(f;..n);\ (0;..b)] \end{array}$$

Now we have to retransform $mkemptyq$.

$$\begin{array}{ll} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge mkemptyq \\ = & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge p'=0 \wedge Q'=Q \quad \text{by several omitted steps} \\ = & m' \ \wedge \ f'=b' \\ \Leftarrow & m:= \top;\ f:= 0;\ b:= 0 \end{array}$$

Now we hope for more success transforming $u:= isemptyq$.

$$\begin{array}{ll} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge (u:= isemptyq) \\ = & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge u'=(p=0) \wedge p'=p \wedge Q'=Q \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by several omitted steps} \\ = & \qquad\quad m \ \wedge \ f<b \ \wedge \ m' \ \wedge \ f'<b' \ \wedge \ b-f = b'-f' \\ & \quad \wedge \ R[f;..b] = R'[f';..b'] \ \wedge \ \neg u' \\ & \vee \quad m \ \wedge \ f<b \ \wedge \ \neg m' \ \wedge \ f'>b' \ \wedge \ b-f = n+b'-f' \\ & \quad \wedge \ R[f;..b] = R'[(f';..n);\ (0;..b')] \ \wedge \ \neg u' \\ & \vee \quad \neg m \ \wedge \ f>b \ \wedge \ m' \ \wedge \ f'<b' \ \wedge \ n+b-f = b'-f' \\ & \quad \wedge \ R[(f;..n);\ (0;..b)] = R'[f';..b'] \ \wedge \ \neg u' \\ & \vee \quad \neg m \ \wedge \ f>b \ \wedge \ \neg m' \ \wedge \ f'>b' \ \wedge \ b-f = b'-f' \\ & \quad \wedge \ R[(f;..n);\ (0;..b)] = R'[(f';..n);\ (0;..b')] \ \wedge \ \neg u' \\ & \vee \quad m \ \wedge \ f=b \ \wedge \ m' \ \wedge \ f'=b' \ \wedge \ u' \\ & \vee \quad \neg m \ \wedge \ f=b \ \wedge \ \neg m' \ \wedge \ f'=b' \\ & \quad \wedge \ R[(f;..n);\ (0;..b)]=R'[(f';..n);\ (0;..b')] \ \wedge \ \neg u' \\ \Leftarrow & u' = (m \wedge f=b) \wedge f'=f \wedge b'=b \wedge R'=R \\ = & u:= \ m \wedge f=b \end{array}$$

The transformed operation offered us the opportunity to rotate the queue within R , but we declined to do so. Each of the remaining transformations offers the same useless opportunity, and we decline each time.

$$\begin{array}{ll} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge (u:= isfullq) \\ = & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge u'=(p=n) \wedge p'=p \wedge Q'=Q \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by several omitted steps} \\ \Leftarrow & u:= \ \neg m \wedge f=b \end{array}$$

$$\begin{array}{ll} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge \mathit{join}\ x \\ = & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge Q'{=}Q[0;..p]{+}x{+}Q[p{+}1;..n] \wedge p'{=}p{+}1 \\ & \qquad\qquad \text{by several omitted steps} \\ \Leftarrow & Rb{:=}\ x;\ \textbf{if}\ b{+}1{=}n\ \textbf{then}\ (b{:=}\ 0;\ m{:=}\ \bot)\ \textbf{else}\ b{:=}\ b{+}1 \end{array}$$

$$\begin{array}{ll} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge \mathit{leave} \\ = & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge Q'{=}Q[(1;..p);(p;..n)] \wedge p'{=}p{-}1 \\ & \qquad\qquad \text{by several omitted steps} \\ \Leftarrow & \textbf{if}\ f{+}1{=}n\ \textbf{then}\ (f{:=}\ 0;\ m{:=}\ \top)\ \textbf{else}\ f{:=}\ f{+}1 \end{array}$$

$$\begin{array}{ll} & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge (u{:=}\mathit{front}) \\ = & \forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge u'{=}Q0 \wedge p'{=}p \wedge Q'{=}Q \\ & \qquad\qquad \text{by several omitted steps} \\ \Leftarrow & u{:=}\ R\,f \end{array}$$

$$\begin{array}{lll} \mathit{Queue2} = & (\mathit{mkemptyq} = m{:=}\top;\ f{:=}\ 0;\ b{:=}\ 0) \\ \quad \wedge & (\mathit{isemptyq} = m \wedge f{=}b) \\ \quad \wedge & (\mathit{isfullq} = \neg m \wedge f{=}b) \\ \quad \wedge & (\mathit{join} = \lambda x{:}\ X\cdot Rb{:=}\ x;\ \textbf{if}\ b{+}1{=}n\ \textbf{then}\ (b{:=}\ 0;\ m{:=}\ \bot) \\ & \qquad\qquad\qquad\qquad \textbf{else}\ b{:=}\ b{+}1) \\ \quad \wedge & (\mathit{leave} = \textbf{if}\ f{+}1{=}n\ \textbf{then}\ (f{:=}\ 0;\ m{:=}\ \top)\ \textbf{else}\ f{:=}\ f{+}1) \\ \quad \wedge & (\mathit{front} = R\,f) \end{array}$$

A transformation can be done by steps, as a sequence of smaller transformations. A transformation can be done by parts, as a conjunction of smaller transformations. But we don't pursue the topic further.

11 Incompleteness

Transformation is sound in the sense that a user cannot tell that a transformation has been made; that was the criterion of its design. But it is possible to find two theories that behave identically from a user's view, but for which there is no transformer to transform one into the other. Transformation is therefore incomplete.

An example to illustrate incompleteness comes from Gardiner and Morgan [3]. The user's variable is i and the implementer's variable is j, both of type nat. The theory is

$$\begin{array}{lll} GM0 & = & (initialize \;=\; i' = 0 \le j' < 3) \\ & \wedge & (step \;=\; \textbf{if } j{>}0 \textbf{ then } (i{:=}\, i{+}1;\; j{:=}\, j{-}1) \textbf{ else } ok) \end{array}$$

The user can look at i but not at j. The user can *initialize*, which starts i at 0 and starts j at any of 3 values. The user can then repeatedly *step* and observe that i increases 0 or 1 or 2 times and then stops increasing, which effectively tells the user what value j started with.

If this were a practical problem, we would notice that *initialize* can be refined, resolving the nondeterminism. For example,

$$initialize \Leftarrow i{:=}\, 0;\; j{:=}\, 0$$

We could then transform *initialize* and *step* to get rid of j, replacing it with nothing. The transformer is $j{=}0$. It transforms the implementation of *initialize* as follows:

$$\begin{array}{ll} & \forall j\cdot j{=}0 \Rightarrow \exists j'\cdot j'{=}0 \wedge i'{=}j'{=}0 \\ = & i{:=}\, 0 \end{array}$$

And it transforms *step* as follows:

$$\begin{array}{ll} & \forall j\cdot j{=}0 \Rightarrow \exists j'\cdot j'{=}0 \;\wedge\; step \\ = & \forall j\cdot j{=}0 \Rightarrow \exists j'\cdot j'{=}0 \;\wedge\; \textbf{if } j{>}0 \textbf{ then } (i{:=}\, i{+}1.\;\; j{:=}\, j{-}1) \textbf{ else } ok \\ = & ok \end{array}$$

The very simple transformed theory

$$\begin{array}{lll} GM1 & = & (initialize \;=\; i{:=}\, 0) \\ & \wedge & (step \;=\; ok) \end{array}$$

cannot be distinguished from the original by the user. If this were a practical problem, we would be done. But the theoretical problem is to replace j with boolean variable b without resolving the nondeterminism, producing the theory

$$\begin{array}{lll} GM2 & = & (initialize \;=\; i'{=}0) \\ & \wedge & (step \;=\; \textbf{if } b \wedge i{<}2 \textbf{ then } i' = i{+}1 \textbf{ else } ok) \end{array}$$

Now *initialize* starts b either at $\top$, meaning that i will be increased, or at $\bot$, meaning that i will not be increased. Each use of *step* tests b to see if we might increase i, and $i<2$ to ensure that i remains below 3. If i is increased, b is again assigned either of its two values. The user will see i start at 0 and increase 0 or 1 or 2 times and then stop increasing, exactly as in the original specification *GM0*. The nondeterminism is maintained. But there is no transformer in variables i, j, and b to do the job; transformation is an incomplete method.

Where there's a will, there's a way. The criterion for being a transformer D is $\forall new\cdot \exists old\cdot D$. This criterion is sufficient to guarantee that when the *old* variables are replaced by the *new*, the result will be correct, but it is not always necessary for correctness. First, we rewrite *GM0* by introducing variable k to stand for the nondeterministically chosen initial value of j.

$$\begin{aligned} GM0 \quad = \quad & \exists k{:}\ 0,..3\cdot\ (initialize \ = \ i{:=}\ 0;\ j{:=}\ k) \\ & \wedge\ (step \ = \ \textbf{if}\ j{>}0\ \textbf{then}\ (i{:=}\ i{+}1;\ j{:=}\ j{-}1)\ \textbf{else}\ ok) \end{aligned}$$

Next we replace j with b using $i{+}j{=}k \wedge b{=}(j{>}0)$. This does not meet the criterion for being a transformer, but it is still safe because $i{+}j{=}k$ is established by *initialize* and maintained invariant by *step*, and $b{=}(j{>}0)$ is a transformer (although it produces an unimplementable result). Now we transform.

$$\begin{aligned} & \forall j\cdot\ i{+}j{=}k \wedge b{=}(j{>}0)\ \Rightarrow\ \exists j'\cdot\ i'{+}j'{=}k \wedge b'{=}(j'{>}0) \wedge initialize \\ = \quad & \forall j\cdot\ i{+}j{=}k \wedge b{=}(j{>}0)\ \Rightarrow\ \exists j'\cdot\ i'{+}j'{=}k \wedge b'{=}(j'{>}0) \wedge i'{=}0 \wedge j'{=}k \\ = \quad & b{=}(i{<}k)\ \Rightarrow\ (i{:=}\ 0;\ b{:=}\ i{<}k) \end{aligned}$$

$$\begin{aligned} & \forall j\cdot\ i{+}j{=}k \wedge b{=}(j{>}0)\ \Rightarrow\ \exists j'\ i'{+}j'{=}k \wedge b'{=}(j'{>}0) \wedge step \\ = \quad & \forall j\cdot \quad i{+}j{=}k \wedge b{=}(j{>}0) \\ & \quad \Rightarrow\ \exists j'\cdot \quad i'{+}j'{=}k \wedge b'{=}(j'{>}0) \\ & \qquad \wedge\ \textbf{if}\ j{>}0\ \textbf{then}\ i'{=}i{+}1 \wedge j'{=}j{-}1\ \textbf{else}\ i'{=}i \wedge j'{=}j \\ = \quad & b{=}(i{<}k)\ \Rightarrow\ b'{=}(i'{<}k) \wedge \textbf{if}\ b\ \textbf{then}\ i'{=}i{+}1\ \textbf{else}\ i'{=}i \\ = \quad & b{=}(i{<}k)\ \Rightarrow\ \textbf{if}\ b \wedge i{<}2\ \textbf{then}\ (i{:=}i{+}1;\ b{:=}\ i{<}k)\ \textbf{else}\ ok \end{aligned}$$

The resulting theory is

$$
\begin{array}{lll}
GM3 = \exists k\colon 0,..3\cdot & & \\
\qquad & (initialize = & b{=}(i{<}k) \Rightarrow (i{:=}\,0;\ b{:=}\,i{<}k)) \\
\quad \wedge\ & (step = & b{=}(i{<}k) \\
\qquad & \Rightarrow & \textbf{if } b \wedge i{<}2 \textbf{ then } (i{:=}i{+}1;\ b{:=}\,i{<}k) \textbf{ else } ok)
\end{array}
$$

Variable j has disappeared, and variable b has appeared, as desired. But we also have k, which we added just to help make the transformation, and now it is no longer wanted. Eliminating it produces *GM2* as desired.

The incompleteness of transformation, like the incompleteness of first-order logic, is demonstrated with an example carefully crafted to show the incompleteness, not one that would ever arise in practice. We should not switch to a more complicated rule, or combination of rules, that are complete. We should stay with the simple rule that is adequate for all transformations that will ever arise in any problem other than a demonstration of theoretical incompleteness. And even then, all we need is to soften the criterion for being a transformer. For further reading, see [7].

12 Conclusion

A theory can be presented as a boolean expression. Theories can then be combined by ordinary conjunction, and by other boolean connectives, and compared for strength by ordinary implication. Strong theories serve mathematicians who want to prove a lot, but weak theories are better for software engineers who need to implement them. This kind of theory presentation is both a simplification and a generalization of the early work. By reducing theories to boolean expressions we understand them in the simplest possible way, and we allow all combinations that make logical sense. Theory refinement is just implication. Implementation can also be expressed as a theory in a particular form. Then implementation is just a refinement, and the proof of correctness of the implementation is just a boolean implication. This kind of theory presentation, as a single boolean expression, works for both the functional style and imperative (state-changing) style of theory.

Theory transformation is a safe and automatic way to reimplement a theory, once the transformer has been written. Although the method of transformation is incomplete in a theoretical sense, it is complete enough for all practical purposes.

References

1. J.-R.Abrial: *the B book, Assigning Programs to Meanings*, Cambridge University Press, 1996

2. R.M.Burstall, J.A.Goguen: "Putting Theories Together to make Specifications", in R.Reddy (ed.): *Proceedings of the fifth International Joint Conference on Artificial Intelligence*, volume 6 pages 1045-1058, Morgan Kaufman , Cambridge MA, 1977

3. P.H.B.Gardiner, C.C.Morgan: "a Single Complete Rule for Data Refinement", *Formal Aspects of Computing*, volume 5 number 4 pages 367-382, 1993

4. J.V.Guttag, J.J.Horning: "the Algebraic Specification of Abstract Data Types", *Acta Informatica*, volume 10 pages 27-52, 1978

5. E.C.R.Hehner: *a Practical Theory of Programming*, first edition Springer 1993, current edition www.cs.utoronto.ca/~hehner/aPToP

6. I.T.Kassios: Theory Theory and an Attempt to Orient Objections to Object Orientation, MSc thesis, University of Toronto, 2001

7. W.-P.deRoever, K.Engelhardt: *Data Refinement: Model-Oriented Proof Methods and their Comparisons*, tracts in Theoretical Computer Science volume 47, Cambridge University Press, 1998

8. J.M.Spivey: *Introducing Z: a Specification Language and its Formal Semantics*, Cambridge University Press, 1988

written 2001, presented at ZB2002 second annual Z and B conference, Grenoble France, 2002 January 23-25