Data refinement of predicate transformers

Paul Gardiner and Carroll Morgan
Programming Research Group, 8-11 Keble Road, Oxford OX1 3QD, U.K.

Communicated by Z. Manna
Received February 1988
Revised February 1989

Abstract

Data refinement is the systematic substitution of one data type for another in a program. Usually, the new data type is more efficient than the old, but also more complex; the purpose of data refinement in that case is to make progress in a program design from more abstract to more concrete formulations. A particularly simple definition of data refinement is possible when programs are taken to be predicate transformers in the sense of Dijkstra. Central to the definition is a function taking abstract predicates to concrete ones, and that function, a generalisation of the abstraction function, therefore is a predicate transformer as well. Advantages of the approach are: proofs about data refinement are simplified; more general techniques of data refinement are suggested; and a style of program development is encouraged in which data refinements are calculated directly without proof obligation.

1. Introduction

In many situations it is simpler to describe the desired result of a task than to describe how a task should be performed. That is particularly true in computer science. Computer programs are very complex, both in their operation and in their representation of information. Yet the task a program performs is often simple to describe.

More confidence in a program's correctness can be gained by describing its intended task in a formal notation. Such specifications can then be used as a basis for a provably correct development of the program. The development can be conducted in small steps, thus allowing the unavoidable complexity of the final program to be introduced in manageable pieces.
The process, called refinement, by which specifications are transformed into programs has received much study in the past. In particular [6, 4, 2, 9] have laid down much of the theory and have recognised two forms of refinement. Firstly algorithmic refinement, where one makes the way in which a program operates more explicit, usually introducing an algorithm where before there was just a statement of the desired result. And secondly data refinement, where one changes the structures for storing information, usually replacing some abstract structure that is easily understood by some more concrete structure that is more efficient.

More recently the emphasis has turned towards providing a uniform theory of program development, in which specifications and programs have equal status. Such a theory is needed to provide the proper setting both for further theoretical work on refinement and for conducting refinement in practice. That goal has been achieved in [2, 13, 10, 11] by extending Dijkstra's language of guarded commands with a specification statement. The extended language, by encompassing both programs and specifications, reduces the process of modular program development to program transformation. Only algorithmic refinement is covered in [13, 10, 11]. In this paper we carry on in the same style to include data refinement, and thus give a more complete framework for software development. In [14], a similar extension is made.

An important part of our approach is the use of predicate transformers, as in [4], which seem to have several advantages over relations, used in [7]. One is that predicate transformers can represent a form of program conjunction not representable in the relational model. That form of conjunction behaves well under data refinement, and can be used to simplify the application of data refinement to specifications. Also, since recursion can be re-expressed in terms of conjunction, that good behaviour allows reasoning about recursion without assuming bounded non-determinism, an unwanted assumption in a theory of programs which includes specifications. Probably the greatest advantage of using predicate transformers, however, is that the theoretical results are so easily applied in practice. In particular, we use a predicate transformer to represent the relationship between abstract and concrete states of data refinement, and that predicate transformer can be used to calculate directly the concrete program from the abstract program. The calculation maintains the algorithmic structure of the program and adds very little extra complication. Moreover, such calculations do not have "applicability conditions": no extra proof of correctness is necessary.

2. Overview

Throughout the paper predicate transformers are used to give meaning to programming and specification language constructs. In Section 3 we introduce the various operators and relations on predicate transformers that are needed for that purpose.

Section 4 gives a short introduction to algorithmic refinement and its relationship to specification.
In Section 5 data refinement is introduced as a transformation on local blocks of a program. It is expressed using a predicate transformer that takes predicates on the abstract state into predicates on the concrete state. It is proven that data refinement applied to a local block has the effect of algorithmically refining the whole program.

In Section 6 Dijkstra's language of guarded commands is extended with constructs for specification. The new constructs are given meaning as predicate transformers, and so have equal status with the other constructs of Dijkstra's language. It is shown that one of these constructs, which we call program conjunction, can be used to formalize the use of logical constants in Hoare-style developments. We also show how Dijkstra's language can be broken up into smaller units than usual.

In Section 7 we prove that data refinement distributes through each construct of the development language. The proofs justify the application of data refinement by parts, retaining the existing algorithmic structure.

In Section 8 a simple method for calculating data refinements of specifications is presented. The calculation is performed by applying a predicate transformer which takes abstract predicates into concrete predicates. It is proven that the calculation yields the most general concrete program.

In Section 9 we discuss the conditions under which fragments of a program can be left unchanged during data refinement. That is important when programs are structured into abstract data types. The conditions justify the use of abstract data types to delineate the scope of data refinement.

In Section 10 a particular predicate transformer for performing data refinement is presented. It has a very simple syntactic form, and is easily applied to the text of a program.

Finally, in Section 11 we give an example of our method applied to a simple program.

3. Predicate transformers

Following [4], we model programs as functions taking predicates to predicates. We intentionally blur the distinction between predicates and the sets of states satisfying them, and therefore we think also of programs as taking sets of (final) states of (initial) states. In any case, for program $P$ and predicate $\psi$, called the *postcondition*, the application of $P$ to $\psi$ is written $P\psi$ and yields a predicate $\phi$, called the *weakest precondition* of $\psi$ with respect to $P$. We say that $P$ transforms $\psi$ into $\phi$. The predicate $\phi$ is the weakest one whose truth *initially* guarantees proper termination of $P$ in a state satisfying $\psi$ *finally*. The expression $P\psi$ can also be read simply as "$P$ establishes $\psi$".

The purpose of predicates in the model is to specify sets of states. For that reason, when giving the meaning of a program as a predicate transformer we will consider only predicates whose free variables are drawn from the program's set of state
variables. We will call the set of state variables the program’s *alphabet* (written $\alpha P$), and call the predicates whose free variables are drawn from a given set of variables $x$ “the predicates on $x$”. Thus, a predicate transformer $P$ can be defined by giving the value of $P\psi$ for all predicates $\psi$ on $\alpha P$. Of course, we have to take care not to apply predicate transformers outside their domains.

For clarity, we will sometimes need to distinguish between program texts and their corresponding predicate transformers, writing $\llbracket T \rrbracket$ for the predicate transformer denoted by the program text $T$.

We define an order $\leq$ on predicates as follows:

$$\phi \leq \psi \iff \models \phi \Rightarrow \psi.$$  

The order $\leq$ permits least upper and greatest lower bounds of collections of predicates $\phi_i$, for which we write, respectively $\bigvee_i \phi_i$ and $\bigwedge_i \phi_i$. Also the order has a top and bottom, $\top$ and $\bot$, which correspond to $\text{true}$ and $\text{false}$.

The order on predicates is promoted to predicate transformers in the usual way; for predicate transformers $P$ and $Q$ such that $\alpha P = \alpha Q$,

$$P \leq Q \iff \text{for all predicates } \phi \text{ on } \alpha P, \ P\phi \leq Q\phi.$$  

The promoted order has least upper and greatest lower bounds as well as a top and bottom element, and they satisfy the following equations:

$$\left( \bigcup_i P_i \right) \phi = \bigvee_i (P_i\phi),$$  

$$\left( \bigcap_i P_i \right) \phi = \bigwedge_i (P_i\phi),$$  

$$\bot \phi = \bot,$$  

$$\top \phi = \top.$$  

All the predicate transformers $P$ we consider are monotonic: for any predicates $\phi$ and $\psi$, $\phi \leq \psi$ implies $P\phi \leq P\psi$.

4. Algorithmic refinement of predicate transformers

In general, one mechanism is refined by another exactly when every specification satisfied by the first is also satisfied by the second. For predicate transformers we take specifications and satisfaction as follows: a *specification* is a pair of predicates comprising the initial assumptions $\text{pre}$ and the final requirement $\text{post}$; and a program $P$ satisfies the specification exactly when

$$\text{pre} \Rightarrow P \text{ post}.$$  

It is now easy to show that $P$ is refined by $Q$ exactly when $P \leq Q$.

The view of specifications distinguishes them from programs. In Section 6, however, we map specifications into the domain of predicate transformers and thus drop the distinction.
5. Data refinement of predicate transformers

During algorithmic refinement, local variables may be introduced. Assignments to local variables do not play a part in the external behaviour of a program. Thus the way local data is stored can be altered without affecting the external properties of a program. Such alterations are called data refinements.

It is common practice for the parts of a program that refer to certain local variables to be named and collected together to form an abstract data type [5]. In Section 9 we discuss data refinement in the context of abstract data types, but here we prefer to work with the most simple structuring mechanism that permits data refinement (i.e., the local declaration).

The following syntax is used to hide (make local) a list of variables x:

\[ [\text{var \ } x \mid I \cdot P] \].

The predicate I states the initialisation of x, and P is the program within which the variables x may be used. The construct is used only if the alphabet of P contains x. The alphabet of the result is that of P with x removed. The meaning of the construct is as follows: for any predicate \( \psi \) on \( (\alpha P - x) \),

\[ \llbracket [\text{var } x \mid I \cdot P] \rrbracket \psi \equiv (\forall x \cdot I \Rightarrow \llbracket P \rrbracket \psi). \]

The definition states that a program with a local variable establishes a predicate if and only if execution of the body, with any initial value of the local variable, establishes the predicate.

We now define data refinement. Let us suppose we wish to replace the list of variables a (the abstract variables) in

\[ [\text{var } a \mid I \cdot P] \]

by some other list of variables \( c \) (the concrete variables), and let the variables of \( \alpha P \), other than a, be \( g \) (the global variables). We choose any predicate transformer \( \text{rep} \) that takes predicates on the variables \( a, g \) to predicates on the variables \( c, g \). Then for programs \( P \) and \( P' \), we write \( P \preceq P' \) to mean that \( P \) is data-refined by \( P' \).

**Definition 5.1.** \( P \preceq P' \) iff \( \text{rep} \circ P \preceq P' \circ \text{rep} \), where the operator \( \circ \) is functional composition (of predicate transformers).

We will see that for data refinement to be well behaved, we must restrict our choice for \( \text{rep} \). We choose \( \text{rep} \) satisfying the following two properties:

- **rep** is monotonic: \( (\forall a \cdot \phi \Rightarrow \psi) \Rightarrow (\forall c \cdot \text{rep } \phi \Rightarrow \text{rep } \psi) \);
- **rep** is \( v \)-distributive: \( \text{rep} \left( \lor \phi_i \right) = \lor \left( \text{rep } \phi_i \right) \).

Note that **strictness** is a special case of **v-distribution** (i.e., \( \text{rep } \bot = \bot \)). Note also that our monotonicity is stronger than the usual. Further properties that follow from
those just defined are proven below. In the proofs that follow and elsewhere we will make use of the fact that the two lists of variables \(a\) and \(g\) are, by definition, disjoint and so the variables \(a\) do not occur free in the predicates on \(g\).

**Lemma 5.2.** If \(\phi\) is a predicate on \(g\), then \(\text{rep } \phi \ll \phi\).

**Proof**

\[
\neg \phi \Rightarrow (\forall a \cdot \phi \Rightarrow \bot) \quad \text{since } a \text{ is not free in } \phi,
\]

\[
\neg \phi \Rightarrow (\text{rep } \phi \Rightarrow \text{rep } \bot) \quad \text{monotonicity of } \text{rep},
\]

\[
\neg \phi \Rightarrow (\text{rep } \phi \Rightarrow \bot) \quad \text{strictness of } \text{rep},
\]

\[
\neg \phi \Rightarrow \neg \text{rep } \phi \quad \text{predicate calculus},
\]

\[
\text{rep } \phi \Rightarrow \phi \quad \text{predicate calculus}. \quad \square
\]

**Lemma 5.3.** If \(\phi\) is a predicate on \(a, g\), and \(\psi\) is a predicate on \(g\), then

\[(\text{rep } \phi) \land \psi \ll \text{rep}(\phi \land \psi).\]

**Proof**

\[
\psi \Rightarrow (\forall a \cdot \phi \Rightarrow \phi \land \psi) \quad \text{since } a \text{ is not free in } \psi,
\]

\[
\psi \Rightarrow (\text{rep } \phi \Rightarrow \text{rep}(\phi \land \psi)) \quad \text{monotonicity of } \text{rep},
\]

\[
(\text{rep } \phi) \land \psi \Rightarrow \text{rep}(\phi \land \psi) \quad \text{predicate calculus}. \quad \square
\]

In subsequent sections we will discuss a particularly convenient choice for \(\text{rep}\), and will show how to calculate suitable \(P'\). For now, we give the fundamental theorem of data refinement.

**Theorem 5.4.** If for suitable \(\text{rep}\) (as defined above) we have shown that \(P \ll P'\), then

\[
[[\text{var } a \mid I \cdot P]] = [[[\text{var } c \mid \text{rep } I \cdot P']].
\]

**Proof.** Let \(\psi\) be any predicate on \(g\); then

\[
[[[\text{var } a \mid I \cdot P]]] \psi = (\forall a \cdot I \Rightarrow [P] \psi) \quad \text{semantics},
\]

\[
\ll (\forall c \cdot \text{rep } I \Rightarrow \text{rep } [P] \psi) \quad \text{monotonicity of } \text{rep},
\]

\[
\ll (\forall c \cdot \text{rep } I \Rightarrow [P'] \text{rep } \psi) \quad \text{hypothesis},
\]

\[
\ll (\forall c \cdot \text{rep } I \Rightarrow [P'] \psi) \quad \text{Lemma 5.2 and monotonicity of } P',
\]

\[
= [[[\text{var } c \mid \text{rep } I \cdot P']] \psi \quad \text{semantics}. \quad \square
\]
6. The development language

The development language is Dijkstra’s language of guarded commands with several extensions. The extensions are to provide a language in which programs and specifications have equal status. Such a language is necessary for the uniform presentation and use of refinement.

All of the predicate transformers \( P \) which can be described by Dijkstra’s original language satisfy the following properties:

- **strictness** \( P \bot = \bot \);
- **monotonicity** \( \phi \leq \psi \) implies \( P\phi \leq P\psi \);
- \( \land \)-**distributivity** \( P(\land \psi_i) = \land_i (P\psi_i) \), for any non-empty family \( \{\psi_i\} \);
- **continuity** \( P(\lor \psi_i) = \lor_i (P\psi_i) \), for any chain \( \{\psi_i\} \).

We will see that some of these properties fail in our extended language.

6.1. Extensions

6.1.1. Local variable declaration

The first extension was given in Section 5; the introduction of local variables \( \var x L \phi P \). It preserves monotonicity and \( \land \)-distributivity, but \( \var x \bot \cdot P \) is not strict for any \( P \), and \( \var n \in \mathbb{N} \cdot m := n \) is not continuous, although \( m := n \) is.

6.1.2. Specification

The second extension is the specification. It includes a pair of predicates, as in Section 4, but here it is added to the development language and given the same status as a programming construct. Specifications are written \( x: [pre, post] \), where \( x \) is a list of the variables that are allowed to change during its execution (\( pre \) and \( post \) are as in Section 4). Specifications are defined as follows:

**Definition 6.1.** \( [x: [pre, post]] \psi \triangleq pre \land (\forall x \cdot post \Rightarrow \psi) \).

The definition states that \( x: [pre, post] \) will establish \( \psi \) if and only if it is executed from a state in which \( pre \) is satisfied and in which values of \( x \) that satisfy \( post \) also satisfy \( \psi \). The specifications of the development language relate to the specifications of Section 4 as follows: a program \( P \) alters only the variables \( x \) and satisfies the specification consisting of predicates \( pre \) and \( post \) if and only if

\[ x: [pre, post] \subseteq P. \]

Specifications are monotonic and \( \land \)-distributive, but \( x: [\top, \bot] \) is not strict, and \( x: [\top, \top] \) is not continuous (take any chain \( \{\psi_i\} \) with \( \lor_i \psi_i = \top \) but \( \psi_i \neq \top \) for any \( i \)).
6.1.3. Program conjunction

The third extension is conjunction of programs, written \( \mathcal{P} \), for any family \( \{P_i\} \), of programs. Its meaning is given as follows

**Definition 6.2.** \( \mathcal{P} \)\( \cong \bigcap \{P_i\} \).

Thus the conjunction of a family of programs refines every member of the family, and is the most non-deterministic program that does so. Conjunction preserves strictness, monotonicity, and continuity, but not \( \land \)-distributivity: consider

\[
[x: [T, \psi] \uparrow x: [T, \neg \psi]][(\psi \land \neg \psi)
\]

for some \( \psi \) not equal to \( \bot \) or \( T \).

Conjunction is an important counterpart of the specification. The specification alone does not allow a relation to be set up between the initial and final states. A similar problem occurs when using Hoare Logic. There, the problem is solved by introducing names for initial values. For example, one might specify that a program \( P \) increments the variable \( n \) by writing

\[
\{n = N\} P \{n = N + 1\},
\]

using \( N \) to denote the initial value of \( n \).

Names such as \( N \) are called logical constants. By convention, they are written in upper case and must not appear in the final program. Since one is required to find \( P \) without knowledge of the value of \( N \), such \( P \) will satisfy the specification for every value.

Those conventions do not work for our specifications, which are treated as programs in their own right and typically appear as fragments of larger programs. For us, \( n: [n = N, n = N + 1] \) is a program with a free variable \( N \). What we need is a program that refines \( n: [n = N, n = N + 1] \) for every value of \( N \). That is where we use program conjunction, writing

\[
\mathcal{P} n: [n = N, n = N + 1],
\]

which is the most non-deterministic such program. A simple calculation confirms that

\[
\mathcal{P} n: [n = N, n = N + 1] = n := n + 1.
\]

Another reason we use the combination of specification and conjunction, other than its relation to logical constants, is that it permits a particularly simple method of calculating data refinements. We will return to data refinement of specifications in Section 8.

That completes the extension of Dijkstra's language. We can now see that the only property retained from the original language is monotonicity: each of the other properties is violated by at least one of the program constructs.
The loss of these properties has no practical disadvantages for us, since we do not prove programs correct by calculating weakest preconditions. Instead, we develop programs stepwise using refinement rules of the form \( P \sqsubseteq Q \), where \( P \) is a specification and \( Q \) is usually a construct such as \( \text{if} \cdots \text{fi} \) surrounding further specifications. A development calculus consisting of rules of that form is presented in [11, 13].

The aim of development is to arrive at a program, written entirely in Dijkstra's original language — the executable subset of our development language. Since every program of that language is strict, the loss of strictness for the general language has some significance. It implies that unimplementable programs can arise, and that developments can run into dead ends. That possibility can be guarded against, if one wishes, by checking the implementability of each new specification that arises in a development, that is by checking

\[
pre \Rightarrow (\exists x \bullet \text{post}).
\]

However, one is not required to perform those checks. Any development that yields an implementation is valid, and the fact that the development calculus does not have the checks built in makes it far simpler than it would be otherwise.

6.2. Generalizations

Having accepted the loss of strictness, continuity and \( \land \)-distributivity, we are able to make other generalizations of the language. Choice and guarding need not be restricted to use within the \( \text{do} \cdots \text{od} \) and \( \text{if} \cdots \text{fi} \) constructs. They can instead be defined as language constructs in their own right.

**Definition 6.3 (Choice).** For any family \( \{P_i\} \) of programs we define their choice as follows:

\[
\llbracket \bigvee_i s_i \rrbracket \triangleq \bigcap_i \llbracket s_i \rrbracket.
\]

Choice preserves monotonicity and \( \land \)-distributivity, but the empty choice is not strict and \( \bigvee_{n \in \text{Nat}} m := n \) is not continuous.

**Definition 6.4 (Guarding).** For predicate \( G \) and program \( P \), we define the guarded command \( G \rightarrow P \) as follows:

\[
\llbracket G \rightarrow P \rrbracket \psi \triangleq G \Rightarrow \llbracket P \rrbracket \psi.
\]

Guarding preserves monotonicity, \( \land \)-distributivity and continuity, but \( \bot \rightarrow P \) is not strict for any \( P \).
Definition 6.5 (Recursion). We must consider program contexts, which are program structures into which program fragments can be embedded. If $c$ is a program context and $P$ a program, then $c(P)$ is a program also. We write $\mu X \cdot c(X)$, where $X$ is a program variable and $c$ is a context, to mean the recursive program consisting of $c$, with $X$ marking the recursive calls. We extend the denotation function $\llbracket \llbracket$ in the obvious way, so that $\llbracket c \rrbracket$ yields a function from predicate transformers to predicate transformers. Thus we have

$$\llbracket \mu X \cdot c(X) \rrbracket \triangleq \text{fix}\llbracket c \rrbracket$$

where fix takes the least fixed point of a function (from predicate transformers to predicate transformers in this case). A least fixed point exists, because each construct of the development language is monotonic with respect to the refinement relation, and so contexts are monotonic also.

Recursion preserves monotonicity, strictness, $\land$-distributivity and continuity.

With these definitions, we can if we wish define the conventional if $\cdots$ fi and do $\cdots$ od constructs as appropriate combinations. We have the following.

Definition 6.6 (Alternation)

\[
\text{if } G_1 \rightarrow P_1 \\
\vdots \\
\Box G_n \rightarrow P_n \\
\text{fi}
\]

is an abbreviation for

\[
\left( \bigwedge_{i=1}^{n} G_i \rightarrow P_i \right) \neg \left( \bigvee_i G_i \right) \rightarrow \text{abort}
\]

Definition 6.7 (Iteration)

\[
\text{do } G_1 \rightarrow P_1 \\
\vdots \\
\Box G_n \rightarrow P_n \\
\text{od}
\]

is an abbreviation for

\[
\left( \mu X \cdot \left( \bigwedge_{i=1}^{n} G_i \rightarrow P_i ; X \right) \right) \llbracket \neg \left( \bigvee_i G_i \right) \rightarrow \text{skip} \rrbracket.
\]
7. Distribution of data refinement

After Theorem 5.4, the most important property of data refinement is that it distributes through the algorithmic constructs of our development language. Only then can one carry over the algorithmic structure of the abstract program onto the concrete program. We prove distribution for each construct below.

**Lemma 7.1 (Sequential composition).** If \( P \preceq P' \) and \( Q \preceq Q' \) then \( P; Q \preceq P'; Q' \).

**Proof**
\[
\begin{align*}
\text{rep} \circ [P; Q] &= \text{rep} \circ [P] \circ [Q] \text{ semantics,} \\
\subseteq [P'] \circ \text{rep} \circ [Q] \text{ hypothesis,} \\
\subseteq [P'] \circ [Q'] \circ \text{rep} \text{ hypothesis and monotonicity of } P', \\
&= [P'; Q'] \circ \text{rep} \text{ semantics. } \square
\end{align*}
\]

**Lemma 7.2 (Skip)** \( \text{skip} \preceq \text{skip} \).

**Proof**
\[
\begin{align*}
\text{rep} \circ [\text{skip}] &= \text{rep} \circ \text{Id} \text{ semantics,} \\
&= \text{rep} \text{ property of Id,} \\
&= \text{Id} \circ \text{rep} \text{ property of Id,} \\
&= [\text{skip}] \circ \text{rep} \text{ semantics. } \square
\end{align*}
\]

**Lemma 7.3 (Abort).** \( \text{abort} \preceq \text{abort} \).

**Proof**
\[
\begin{align*}
\text{rep} \circ [\text{abort}] &= \text{rep} \circ \bot \text{ semantics,} \\
&= \bot \text{ strictness of rep,} \\
&= \bot \circ \text{rep} \text{ property of } \bot, \\
&= [\text{abort}] \circ \text{rep} \text{ semantics. } \square
\end{align*}
\]

**Lemma 7.4 (Guarding).** To deal with guarded commands we will need another function from abstract predicates to concrete predicates.
\[
\text{rep } \psi \equiv \neg (\text{rep } \neg \psi).
\]

We then have the following result for guarded commands. If \( P \preceq P' \) then \((G \rightarrow P) \preceq ((\text{rep } G) \rightarrow P')\).
Proof

\[ \text{rep}[G \rightarrow P] \psi \]
\[ = \text{rep}(G \Rightarrow \Box P) \psi \]
\[ = \text{rep}(\neg G \lor \Box P) \psi \]
\[ = (\text{rep} \neg G \lor (\text{rep} \Box P) \psi) \]
\[ = \neg (\text{rep} G) \lor (\text{rep} \Box P) \psi \]
\[ = (\text{rep} G') \Rightarrow (\text{rep} \Box P) \psi \]
\[ = \text{semantics}, \text{predicate calculus}, \lor -\text{distributivity of rep}, \text{definition of rep}, \text{predicate calculus}, \text{hypothesis}, \text{semantics}. \]

Lemma 7.5 (Choice). If for each \( i \), \( P_i \leq P'_i \), then \( \Box \leq \Box, P_i \leq \Box, P'_i \).

Proof

\[ \text{rep} \circ \left[ \bigwedge_i \Box P_i \right] \]
\[ = \text{rep} \circ \left( \bigwedge_i \Box P_i \right) \]
\[ = \text{monotonicity of rep}, \text{hypothesis}, \text{property of \( \Box \)}, \text{semantics}. \]

Lemma 7.6 (Conjunction). If for each \( i \), \( P_i \leq P'_i \), then \( \bigwedge_i P_i \leq \bigwedge_i P'_i \).

Proof

\[ \text{rep} \circ \left[ \bigvee_i \Box P_i \right] \]
\[ = \text{rep} \circ \left( \bigvee_i \Box P_i \right) \]
\[ = \lor -\text{distributivity of rep}, \text{hypothesis}, \text{property of \( \bigvee \)}, \text{semantics}. \]
Lemma 7.7 (Recursion). We first promote data refinement to program contexts: we say that $\mathcal{C} \leq \mathcal{C}'$ exactly when for all pairs of programs $P$ and $P'$ such that $P \leq P'$, we have $\mathcal{C}(P) \leq \mathcal{C}'(P')$ as well. The result for recursion is then as follows:

If $\mathcal{C} \leq \mathcal{C}'$, then $(\mu X \cdot \mathcal{C}(X)) \leq (\mu X \cdot \mathcal{C}'(X))$.

Proof. The Limit Theorem (as generalized by Hitchcock and Park) asserts that, for monotonic $F$, there exists an ordinal $\lambda$ such that $\text{fix } F = F^\lambda \perp$, where

$F^0 X = X,$

$F^\alpha+1 X = F(F^\alpha X),$

$F^\lambda X = \bigsqcup_{\beta < \lambda} (F^\beta X).$

Hence, it is sufficient to prove $[\mathcal{C}]^\lambda \perp \leq [\mathcal{C}']^\lambda \perp$ for all $\lambda$. That can be proven by induction. The base case follows from Lemma 7.3, the step case follows from $\mathcal{C} \leq \mathcal{C}'$ and the limit case follows from Lemma 7.6. $\Box$

8. Data refinement of specifications

In the preceding section, we showed that data refinement can be performed piecewise (a term we borrow from [14]), thus maintaining the algorithmic structure of a program. We now consider the pieces lying within the structure.

There are two constructs to consider, the specification and the assignment. In fact, we can ignore the assignment statement since it is readily transformed into a simple specification. The following theorem provides a method for calculating the result of applying data refinement to a specification.

Again, we write $a$ for the list of abstract variables, $c$ for the list of concrete variables, $g$ for the remaining (global) variables and $\text{rep}$ for the representation predicate transformer.

Theorem 8.1. $g, a: [\text{pre, post}] \leq g, c: [\text{rep pre, rep post}]$.

Proof. Let $\psi$ be any predicate on $g, a$; then

$\text{rep}[g, a: [\text{pre, post}]]\psi$

$= \text{rep}(\text{pre} \land (\forall g, a \cdot \text{post} \Rightarrow \psi))$ semantics,

$\leq (\text{rep pre}) \land \text{rep}(\forall g, a \cdot \text{post} \Rightarrow \psi)$ monotonicity of rep,

$\leq (\text{rep pre}) \land (\forall g, a \cdot \text{post} \Rightarrow \psi)$ Lemma 5.2,

$\leq (\text{rep pre}) \land (\forall g, \text{c } \cdot \text{rep post} \Rightarrow \text{rep } \psi)$ monotonicity of rep,

$= [g, c: [\text{rep pre, rep post}]]\text{rep } \psi$ semantics. $\Box$
Data refinement may partly resolve the non-determinism exhibited by the abstract program. We would like our calculating of data refinements to leave as much choice as possible for later algorithmic refinement steps. Thus we would like the calculation to produce the concrete program that is least in the refinement ordering. That is confirmed by the following theorem.

**Theorem 8.2.** If \( g, a: [\text{pre}, \text{post}] \leq P \), then
\[
 g, c: [\text{rep pre}, \text{rep post}] \leq P.\]

**Proof.** Let \( \psi \) be any predicate on \( g, c \); then
\[
\semantics{\semantics{g, c: [\text{rep pre}, \text{rep post}]}(\psi)}
= (\text{rep pre}) \land (\forall g, c \cdot \text{rep post} \Rightarrow \psi) \quad \text{semantics},
\]
\[
\leq (\text{rep pre}) \land (\forall g, a \cdot \text{post} \Rightarrow \bigvee_{g, c \cdot \text{rep x} = \psi} x) \quad \text{properties of } \bigvee,
\]
\[
\leq \text{rep}\left(\text{pre} \land (\forall g, a \cdot \text{post} \Rightarrow \bigvee_{g, c \cdot \text{rep x} = \psi} x)\right) \quad \text{Lemma 5.3},
\]
\[
= \text{rep}\left(\semantics{g, a: [\text{pre}, \text{post}]} \bigvee_{g, c \cdot \text{rep x} = \psi} x\right) \quad \text{semantics},
\]
\[
\leq P\left(\bigvee_{g, c \cdot \text{rep x} = \psi} x\right) \quad \text{hypothesis},
\]
\[
= P\left(\bigvee_{g, c \cdot \text{rep x} = \psi} \text{rep x}\right) \quad \text{v-distributivity of } \text{rep},
\]
\[
\leq P\psi \quad \text{properties of } \bigvee \text{ and } \text{monotonicity of } P.
\]

Theorem 8.1 generalizes so that \( g \) can be any subset of the global variables, but then it is not necessarily the most non-deterministic concrete program that is calculated.

9. Abstract data types

So far we have been considering the application of data refinement to the whole of a local block. However, when a block is structured using abstract data types it is broken up and the parts become separated. The parts that refer to the local variables are collected together to form the abstract data type, leaving in place the parts that make no mention of the local variables.

With that type of structuring it is essential that we are able to apply data refinement to the code fragments that make up the abstract data type, without having to consider the other parts of the local block. Even when a program is not structured in that way, it is still convenient to be able to ignore fragments of code that do not mention
the variables being refined. Thus we need to show that $P \preceq P$ holds for all programs $P$ which do not refer to abstract variables.

Most constructs of the development language do not introduce references to variables, so we need consider only guards and specifications. For guards, the proof is trivial, but for specifications, a further property of $\mathit{rep}$ is required.

**Definition 9.1.** $\mathit{rep}$ is local if it maps predicates on $a$ into predicates on $c$.

Hence, if $\mathit{rep}$ is local and $g$ does not occur free in $\psi$, then $g$ does not occur free in $\mathit{rep}\psi$. From that property of $\mathit{rep}$, the desired result for specifications follows. First we prove a lemma.

**Lemma 9.2.** If $\mathit{rep}$ is local, then

\[ \mathit{rep}(\forall g \cdot \psi) \preceq (\forall g \cdot \mathit{rep}\psi). \]

**Proof.**

\begin{align*}
(\forall g \cdot \psi) \Rightarrow \psi & \quad \text{predicate calculus,} \\
\mathit{rep}(\forall g \cdot \psi) \Rightarrow \mathit{rep}\psi & \quad \text{monotonicity of } \mathit{rep}, \\
\mathit{rep}(\forall g \cdot \psi) \Rightarrow (\forall g \cdot \mathit{rep}\psi) & \quad \text{$g$ not free in } \mathit{rep}(\forall g \cdot \psi) \\
& \quad \text{by locality.} \quad \Box
\end{align*}

**Theorem 9.3.** If $\mathit{rep}$ is local and the variables $a$ do not occur free in $\mathit{pre}$ and $\mathit{post}$, then

\[ g: [\mathit{pre}, \mathit{post}] \preceq g: [\mathit{pre}, \mathit{post}]. \]

**Proof.** Let $\psi$ be any predicate on $g, a$; then

\[ \mathit{rep}[g: [\mathit{pre}, \mathit{post}]]\psi \]

\[ = \mathit{rep}(\mathit{pre} \land (\forall g \cdot \mathit{post} \Rightarrow \psi)) \quad \text{semantics,} \]

\[ \preceq (\mathit{rep}\mathit{pre}) \land \mathit{rep}(\forall g \cdot \mathit{post} \Rightarrow \psi) \quad \text{monotonicity of } \mathit{rep}, \]

\[ \preceq \mathit{pre} \land \mathit{rep}(\forall g \cdot \mathit{post} \Rightarrow \psi) \quad \text{Lemma 5.2,} \]

\[ \preceq \mathit{pre} \land (\forall g \cdot \mathit{rep}(\mathit{post} \lor \psi)) \quad \text{Lemma 9.2,} \]

\[ = \mathit{pre} \land (\forall g \cdot \mathit{rep}(\lnot\mathit{post} \lor \psi)) \quad \text{predicate calculus,} \]

\[ = \mathit{pre} \land (\forall g \cdot \mathit{rep}(\lnot\mathit{post} \lor \mathit{rep}\psi)) \quad \lor\text{-distributivity of } \mathit{rep}, \]

\[ \preceq \mathit{pre} \land (\forall g \cdot \lnot\mathit{post} \lor \mathit{rep}\psi) \quad \text{Lemma 5.2,} \]

\[ = \mathit{pre} \land (\forall g \cdot \mathit{post} \Rightarrow \mathit{rep}\psi) \quad \text{predicate calculus,} \]

\[ = [g: [\mathit{pre}, \mathit{post}]]\mathit{rep}\psi \quad \text{semantics.} \quad \Box \]

10. Data refinement in practice

So far we have given no indication as to how one chooses a suitable representation transformer $\mathit{rep}$. In fact, there may be many classes of program transformation that
can be supported by the theory of the preceding sections. We can, though, cite one example that proves very useful in practice. The definition of $rep$ given below provides the same form of data refinement as that in [2, 5, 14] and also, under the name of downward simulation, in [8].

When performing data refinement, one always intends that the abstract, concrete and global states should correspond in some way. The correspondence can be expressed as a predicate over the three sets of state variables. From such a predicate ($I$, say) the representation transformer can be defined as follows:

$$rep \phi \triangleq (\exists a \cdot I \land \phi).$$

If is easy to verify the properties required for data refinement (i.e., monotonicity and $\vee$-distributivity). If $I$ does not refer to the global variables then $rep$ also has the property required for applying data refinement to abstract data types (i.e., locality).

$rep$ also has a simple form:

$$\overline{rep} \phi = (\forall a \cdot I \Rightarrow \phi).$$

With our choice of $rep$, we can see how easily data refinement can be performed by calculation. The two simple transformers, $rep$ and $\overline{rep}$ can be applied directly to the predicates of any abstract program to calculate a concrete refinement. In the abstract program all the pre and post conditions of specifications are replaced by their image under $rep$ and the guards by their image under $\overline{rep}$, thus leaving the algorithmic structure unchanged. The result is guaranteed correct by the theorems of the previous sections.

11. Example

A typical development would alternate between algorithmic and data refinement. For this example, we will consider a development in which some algorithmic refinement has already occurred, thus producing the following program:

$$\begin{align*}
  | & \{ \textbf{var } b \mid b = \{ \} \} \cdot \\
  & \vdots \\
  & \vdash b : [b = B, b = B \cup \{e\}] \\
  & \vdots \\
  & \vdash n, b : \left[ b = B, \begin{array}{l}
    b = B \\
    n = \max B
  \end{array} \right] \\
  & \vdots \\
  & ].
\end{align*}$$
Only the fragments that refer to the local variable $b$ have been displayed. The variable $b$ holds a set of natural numbers, and is initially empty. The first specification describes the addition, to the set, of an element $e$. The second describes the setting of the variable $n$ to the maximum of the set.

The data refinement we wish to conduct is that of replacing the set $b$ by a number $m$ that records the maximum of the set. The relationship between the two variables is

$$ m = \text{max } b. $$

Hence the representation transformer is defined as follows:

$$ \text{rep } \psi \triangleq (\exists b \cdot m = \text{max } b \land \psi). $$

We can apply the transformer directly to the abstract program, and thus produce the concrete program:

$$ \{ \text{var } m | (\exists b \cdot b = \{ \} \land m = \text{max } b) \} \star $$

$$ \vdots $$

$$ \begin{align*}
\vdash m & \vdash \left( (\exists b \cdot b = B \land m = \text{max } b) , (\exists b \cdot b = B \cup \{ e \} \land m = \text{max } b) \right) \\
\vdash n, m & \vdash \left( (\exists b \cdot b = B \land m = \text{max } b) , (\exists b \cdot n = \text{max } B \land m = \text{max } b) \right)
\end{align*} $$

Using the predicate calculus (one-point law), that simplifies to

$$ \{ \text{var } m | m = \text{max } \{ \} \} \star $$

$$ \vdots $$

$$ \begin{align*}
\vdash m & \vdash [m = \text{max } B, m = \text{max } (B \cup \{ e \})] \\
\vdash \vdash n, m & \vdash [m = \text{max } B, n = \text{max } B] \\
\vdash \vdash n, m & \vdash [m = \text{max } B, m = \text{max } B]
\end{align*} $$

The initialization further simplifies to $m = 0$, and the first specification simplifies to

$$ m : [m = \text{max } B, m = \text{max } \{ \text{max } B, e \}]. $$
Now algorithmic refinement can continue. We can apply the assignment introduction rule [11], which asserts that if \( pre \Rightarrow post[x\ E] \), then

\[
x: [pre, post] \sqsubseteq x := E,
\]

where \( post[x\ E] \) means the result of substituting \( E \) for \( x \) in \( post \). Hence we can refine the first specification to \( m := \max\{m, e\} \).

The same refinement rule can be applied to the second specification to produce the assignment \( n := m \). Lastly, since the logical constant \( B \) is no longer referred to, the program conjunctions can be dropped, leaving us with the following program:

\[
[[\text{var} m]m = 0\ .

\ldots

m := \max\{m, e\}

\ldots

n := m

\ldots

]].

12. Conclusions

We have presented the familiar technique of data refinement in the context of predicate transformers. In doing so we have drawn on other recent work in program development: the factoring of Dijkstra's language into smaller pieces (Definitions 6.6 and 6.7); the use of recursion in practice rather than iteration as the basis for unbounded computations in Dijkstra's language [3]; and the mixing of specification and program [2, 13, 10, 11]. In Definition 6.2 we give a further factorization: with program conjunction, the technique of "logical" constants is formalized. All of that comes together to promote a style of program design in which steps are made by calculation rather than via proof obligations.

It is clear, though, that proof cannot be avoided altogether! In practice, the necessary truths of predicate calculus are drawn on when strengthening postconditions and weakening preconditions — and virtually nowhere else. In that respect perhaps the proofs have moved rather than disappeared. Their confinement though makes the other rules easier to apply in practice. Certainly they are easier to remember: the formulation of data refinement in Theorem 8.1 is simpler than any other of which we are aware.

Using predicate transformers as a model, rather than the relations of [8], has several advantages. One is that the dependence on continuity is more easily broken. This allowed us to extend the work of [8] so that it applies to a language that includes constructs for specification as well as programming. Another advantage is the ease with which the theoretical results are applied in practice. Both those
advantages are related to program conjunction, which cannot be represented in the relational model. Conjunction simplifies the expression of specifications in our language; and that, in turn, permits our very simple method of data refinement calculation. In contrast, the method of calculation in [8], although theoretically simple, gives rise to very large and unwieldy expressions in practice.

Our relaxing of Dijkstra’s “healthiness” conditions has left us only with monotonicity: continuity, strictness, and ∧-distributivity are gone. That is similar to [13], where continuity and strictness are dropped so that the guard and choice symbols can be given meaning as operators in their own right. We too proposed that in [10], but have taken the process further, dropping also ∧-distributivity, so that we can define the conjunction operator which is the key to simplifying the calculation of data refinements.

Our results are potentially more general than those of [14], since we recognise how the abstraction condition is itself applied as a predicate transformer, and base all our proofs on three properties of it. By doing that we make the structure of the proofs more explicit, and we leave open the possibility of finding other predicate transformers with those properties which can, therefore, also be used for data refinement.

We also prove results that [14] does not. Theorem 5.4 forms the important link between data refinement of local variables and operational refinement. Theorem 8.2 shows that our method of calculation yields the weakest program that is a data refinement of the original and thus that no loss of choice is incurred by calculation.

Acknowledgment

An early description of data refinement appeared in [6], and was later made part of the Vienna Development Method [9]. Specifications were embedded within programs in [2], where data refinement was also treated. The first connection between data refinement and weakest preconditions was made by [2], though it had been earlier presented by [4] as a technique based on auxiliary variables. ([12] explains the connection between these.) Most recently, [14] has given the same formulation as Section 5.

Our own work owes much to collaboration with Jean-Raymond Abrial who has been a constant source of new and exciting ideas. We would like to thank the referees for pointing out several errors and for making many helpful suggestions. We would also like to thank British Petroleum for their financial support.

References