Conjoining Derivations of Loops and Recursions in the Refinement Calculus

Lindsay Groves

School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand
lindsay@mcs.vuw.ac.nz

Abstract

We review the approach to adapting program derivations, based on properties of a program conjunction operator, presented in [Groves, 1998]. The law for distributing conjunction over loops is extended to provide a more general law for combining recursion blocks with common structure.

1 Introduction

In [Groves, 1998], we discuss the problem of adapting a program derivation to obtain an implementation for a modified specification. We introduce a conjunction operator that can be used to combine programs (including specifications) in a wide spectrum language, and argue that this can be used to express some of the kinds of modifications to specifications that arise during incremental program development and program maintenance. We investigate a number of properties of this operator and present a number of laws that can be used to propagate a specification change through an existing derivation in order to obtain an implementation for the new specification. The ideas are presented in the context of the refinement calculus (e.g. [Morgan, 1994]), so specifications are usually specification statements, and derivations are sequences of refinement steps.

One of the most significant laws presented in [Groves, 1998] is a law for distributing conjunction over repetition, which allows a conjunction of two loops to be transformed into a single loop. This laws seems (to me, at least) surprisingly simple and had not been proved rigorously at that point, although all attempts to a find a counter-example had failed. In this paper, we present a more general law for distributing conjunction over recursion blocks, from which the repetition law presented in [Groves, 1998] follows as a special case.

We begin by giving a brief description of the program conjunction operator and the properties and laws needed in this paper. We then introduce recursion blocks and give the new law for distributing conjunction over recursion blocks, show how the loop law in [Groves, 1998] is obtained as a special case, and give some examples. We then conclude.

2 Program Conjunction

For two programs (or specifications) S and T, we write $S \circledcirc T$ to denote the conjunction of S and T. We think of $S \circledcirc T$ as saying, roughly, "Do what S says and what T says". More precisely, $S \circledcirc T$ is a program that will have the effect of S and (simultaneously) the effect of T, provided that both S and T can be executed. This can be viewed as a kind of parallel composition, where S and T are executed in parallel and their results combined.

For example, writing $\#A$ for the length of sequence A, $\sqcap A$ for its minimum, and $x\#A$ for the number of occurrences of x in A, we may wish to combine specifications

$$lo:\left[\#A \geq 1 \;\middle/\; lo = \sqcap A\right] \quad \text{and} \quad c:\left[\#A \geq 0 \;\middle/\; c = lo\#A\right]$$

to obtain a specification describing a program to find the minimum (lo) of A and the number of times it occurs (c). Note that counting is defined when $\#A \geq 0$, while finding the minimum requires $\#A \geq 1$ — we can only do both when $\#A \geq 1$.

In this paper, we will assume that S and T denote conjunctive, monotonic predicate transformers over a common frame. This means that the combined specification outlined above would be written as:

$$lo, c:\left[\#A \geq 1 \;\middle/\; lo = \sqcap A\right] \circledcirc lo, c:\left[\#A \geq 0 \;\middle/\; c = lo\#A\right]$$

The above description of $\circledcirc$ suggests that the following properties are required:

$$\begin{aligned} \{\, pre(T) \,\}\, S &\sqsubseteq S \circledcirc T \\ \{\, pre(S) \,\}\, T &\sqsubseteq S \circledcirc T \end{aligned}$$

When S and T are specification statements, it is easy to see that the least refined conjunctive solution to these constraints is given by:

$$w:\left[P_1 \;\middle/\; R_1\right] \circledcirc w:\left[P_2 \;\middle/\; R_2\right] \;\equiv\; w:\left[P_1 \wedge P_2 \;\middle/\; R_1 \wedge R_2\right]$$

Since any specification, $w:\left[P \;\middle/\; R\right]$, can be written as an assumption ($\{P\}$), followed by an unconstrained assignment ($w := ?$, sometimes called **choose**), followed by a coercion ($[R]$), we also have:

$$w:\left[P_1 \;\middle/\; R_1\right] \circledcirc w:\left[P_2 \;\middle/\; R_2\right] \;\equiv\; \{\, P_1 \wedge P_2 \,\} \;;\; w := ? \;;\; [\, R_1 \wedge R_2 \,]$$

The conjunction operator discussed in [Groves, 1998] does not assume identical frames, but instead (following [Mahony, 1998]) indicates the variables of concern to each of the programs being conjoined. Assuming identical frames avoids a number of technical problems, but does not otherwise significantly affect the results presented here. It does, however, mean that the statements of several of the properties of conjunction, and the conditions applying to the laws, are somewhat different. This assumption and its consequences are discussed further in Section 2.2.

This conjunction operator, or ones very similar, are defined in [Utting, 92], [Ward, 93], [Ainsworth and Wallis, 94] and [Back and Butler, 1996]. Ward calls the operator *miraculous conjunction*, and introduces it as a step towards defining a Z-like conjunction operator in the refinement calculus. A comparison of these and other related operators is given in [Mahony, 1998].

A significant difference between this conjunction operator and the conjunction operators found in Z and various other relational settings (e.g. [Hehner, 1993], [Hoare *et al*, 1987]) is that this operator is monotonic — it becomes miraculous if the programs being conjoined are inconsistent. Rather than expanding conjunctive specifications using the above relationship before implementing the specification, we want to be able to (partially) refine the conjoined specifications separately and then distribute the conjunction over the resulting programs. If we already have a derivation for S, and T is an added requirement which can be satisfied by modifying the existing program (as in the above example), we attempt to mimic the refinement steps of the original derivation on the additional specification, and then use distribution laws to merge the programs thus obtained. It is thus important for us that the conjunction operator be monotonic. The fact that conjoining inconsistent specifications results in magic also turns out to be advantageous, since it simplifies the proof obligations for many of the laws.

We can extend the above definition to define $S \owedge T$ for arbitrary programs S and T, by defining generalised precondition and effect operators (see [Mahony, 1998]). We will bypass the detail involved by noting that any conjunctive predicate transformer can be expressed as a specification statement, provided that infinitary formulae are allowed [Utting, 92].

In the rest of this section, we present some basic properties of program conjunction and explain the ramifications of our assumption that programs being conjoined have identical frames. More details can be found in [Groves, 1998].

2.1 Basic Properties of Program Conjunction

The following are some of the basic properties of program conjunction. We assume that S and T have frame w, and that the precondition of a program is given by $pre(\_)$.

- Conjunction is monotonic with respect to refinement.

$\owedge$*-mono:* $\quad S \owedge T \;\sqsubseteq\; S' \owedge T', \quad \text{if } S \;\sqsubseteq\; S' \text{ and } T \;\sqsubseteq\; T'$

- Conjunction is idempotent, commutative and associative.

$\owedge$*-idem:* $\quad S \owedge S \;\equiv\; S$

$\owedge$*-comm:* $\quad S \owedge T \;\equiv\; T \owedge S$

$\owedge$*-assoc:* $\quad (S \owedge T) \owedge U \;\equiv\; S \owedge (T \owedge U)$

- Certain conjunctions involving atomic statements can be eliminated.

$$\text{⊗-}\mathbf{abort}\text{:}\quad \mathbf{abort} \circledwedge S \;\equiv\; \mathbf{abort}$$

$$\text{⊗-}\mathbf{magic}\text{:}\quad \mathbf{magic} \circledwedge S \;\equiv\; \{\, pre(S) \,\} \;;\; \mathbf{magic}$$

$$\text{⊗-}\mathbf{choose}\text{:}\quad w := ? \circledwedge S \;\equiv\; S$$

- Conjunctions of assertions, or of coercions, can be merged.

$$\text{⊗-}\textit{assert:}\quad \{\, P_1 \,\} \circledwedge \{\, P_2 \,\} \;\equiv\; \{\, P_1 \wedge P_2 \,\}$$

$$\text{⊗-}\textit{coerce:}\quad [R_1] \circledwedge [R_2] \;\equiv\; [R_1 \wedge R_2]$$

- Conjunctions of assignments can be eliminated; inconsistent assignments to the same variable lead to **magic**.

$$\begin{aligned} \text{⊗-}\textit{asn-1:}\quad w := e \circledwedge w := f \;&\equiv\; [e = f]\; w := e \\ &\equiv\; \begin{cases} w := e, & \text{when } e = f \\ \mathbf{magic}, & \text{when } e \neq f \end{cases} \end{aligned}$$

2.2 Frames

Given a program $S \equiv w \mathbin{:}\!\left[P \;/\; R\right]$ with frame w, we can extend S to obtain a program S' on a larger frame $w \cup x$ in either of two ways:

Restrictive extension The implementation of S' cannot change x:

$$S' \;\equiv\; w, x \mathbin{:}\!\left[P \;/\; R \wedge x = x_0\right]$$

Unrestrictive extension The implementation of S' may change x:

$$S' \;\equiv\; w, x \mathbin{:}\!\left[P \;/\; R\right]$$

In forming the conjunction $S \circledwedge T$ of programs S and T, we need to ensure that the conjunction is able to modify all variables that can be modified by either S or T. We therefore need to take the unrestrictive extensions of S and T, with respect to variables in the frame of one but not the other, rather than the usual restrictive extension.

Thus, when we conjoin $w_1 \mathbin{:}\!\left[P_1 \;/\; R_1\right]$ and $w_2 \mathbin{:}\!\left[P_2 \;/\; R_2\right]$, where $w_1 \neq w_2$, we get $w \mathbin{:}\!\left[P_1 \;/\; R_1\right] \circledwedge w \mathbin{:}\!\left[P_2 \;/\; R_2\right]$, where $w = w_1 \cup w_2$.

For example, conjoining $x := e$ and $y := f$, for distinct x and y, gives $x, y := e, ? \circledwedge x, y := ?, f$, which is equivalent to $x, y := e, f$.

3 Distributing Conjunctions

Given a conjunction $S \owedge T$, where S and T are both formed with sequential composition, alternation or variable declaration, we can distribute the conjunction over the program constructor. Here we only give the laws needed in the treatment of recursion and loops, and combine them to give a more general law; [Groves, 1998] gives various special cases of these laws.

3.1 Sequence

When S and T are both sequential compositions, we can distribute the conjunction over both sides, synchronising at their mid-points. If any interference is introduced, the result is magic. Thus, a poor choice of mid-point will lead to infeasibility, but never to an incorrect program![1]

$\owedge$*-dist-seq:* $(S_1 \;;\; S_2) \owedge (T_1 \;;\; T_2) \;\sqsubseteq\; (S_1 \owedge T_1) \;;\; (S_2 \owedge T_2)$

Proof Let $S_1 \equiv w\mathbin{:}\left[P_1 \;\middle/\; Q_1\right]$, $S_2 \equiv w\mathbin{:}\left[Q_1 \;\middle/\; R_1\right]$, $T_1 \equiv w\mathbin{:}\left[P_2 \;\middle/\; Q_2\right]$ and $T_2 \equiv w\mathbin{:}\left[Q_2 \;\middle/\; R_2\right]$.

$$
\begin{array}{ll}
& (S_1 \;;\; S_2) \owedge (T_1 \;;\; T_2) \\[1ex]
\equiv & w\mathbin{:}\left[P_1 \;\middle/\; Q_1\right] ; w\mathbin{:}\left[Q_1 \;\middle/\; R_1\right] \;\owedge\; w\mathbin{:}\left[P_2 \;\middle/\; Q_2\right] ; w\mathbin{:}\left[Q_2 \;\middle/\; R_2\right] \\[1ex]
\equiv & w\mathbin{:}\left[P_1 \;\middle/\; (\exists w' \bullet Q_1[w \backslash w'] \wedge R_1[w_0 \backslash w'])\right] \owedge \\
& w\mathbin{:}\left[P_2 \;\middle/\; (\exists w' \bullet Q_2[w \backslash w'] \wedge R_2[w_0 \backslash w'])\right] \\[1ex]
\equiv & w\mathbin{:}\left[P_1 \wedge P_2 \;\middle/\; \begin{array}{c}(\exists w' \bullet Q_1[w \backslash w'] \wedge R_1[w_0 \backslash w']) \wedge \\ (\exists w' \bullet Q_2[w \backslash w'] \wedge R_2[w_0 \backslash w'])\end{array}\right] \\[1ex]
\sqsubseteq & w\mathbin{:}\left[P_1 \wedge P_2 \;\middle/\; (\exists w' \bullet (Q_1 \wedge Q_2)[w \backslash w'] \wedge (R_1 \wedge R_2)[w_0 \backslash w'])\right] \\[1ex]
\equiv & w\mathbin{:}\left[P_1 \wedge P_2 \;\middle/\; Q_1 \wedge Q_2\right] \;;\; w\mathbin{:}\left[Q_1 \wedge Q_2 \;\middle/\; R_1 \wedge R_2\right] \\[1ex]
\equiv & (S_1 \owedge T_1) \;;\; (S_2 \owedge T_2)
\end{array}
$$

□

[1] This is one of the advantages of our (miraculous) conjunction over a Z-like conjunction.

3.2 Alternation

When S and T are both alternations, with identical guard sets, we can distribute the conjunction over both alternations.[2]

$$\text{\textcircled{$\wedge$}-}\textit{dist-alt:}\quad \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow S_i\ \textbf{fi}\ \textcircled{$\wedge$}\ \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow T_i\ \textbf{fi}$$
$$\equiv\ \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow S_i \textcircled{$\wedge$} T_i\ \textbf{fi}$$

Proof We first show that $\textbf{if}\ \left[\!\right]_i\ G_i \rightarrow S_i\ \textbf{fi} \textcircled{$\wedge$} T\ \equiv\ \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow S_i \textcircled{$\wedge$} T\ \textbf{fi}$.
Assuming $T \equiv w\!:\!\left[P\ \middle/\ R\right]$, we have:

$$\begin{aligned}
& \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow S_i\ \textbf{fi}\ \textcircled{$\wedge$}\ T \\
\equiv\ & \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow S_i\ \textbf{fi}\ \textcircled{$\wedge$}\ w\!:\!\left[P\ \middle/\ R\right] \\
\equiv\ & \{\,P\,\}\ ;\ \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow S_i\ \textbf{fi}\ ;\ w\!:\!\left[R\right] \\
\equiv\ & \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow \{\,P\,\}\,;S_i\ ;w\!:\!\left[R\right]\textbf{fi} \\
\equiv\ & \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow S_i \textcircled{$\wedge$} w\!:\!\left[P\ \middle/\ R\right]\textbf{fi} \\
\equiv\ & \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow S_i \textcircled{$\wedge$} T\ \textbf{fi}
\end{aligned}$$

We now obtain the required result by taking $T \equiv \textbf{if}\ \left[\!\right]_i\ G_i \rightarrow T_i\ \textbf{fi}$, and applying this result again to each $S_i \textcircled{$\wedge$} T$, and flattening the resulting nested alternations.
□

3.3 Variable declarations

When S and T are both variable declarations, we can distribute the conjunction over the two declaration blocks, forcing the combined program to use the same local variable.

$$\text{\textcircled{$\wedge$}-}\textit{dist-var:}\quad [\![\ \textbf{var}\ x \bullet S'\ ]\!] \textcircled{$\wedge$} [\![\ \textbf{var}\ x \bullet T'\ ]\!]\ \sqsubseteq\ [\![\ \textbf{var}\ x \bullet S' \textcircled{$\wedge$} T'\ ]\!]$$

[2]This can be easily extended to allow different guard sets, but that version is not required here.

Proof First, we observe that, for disjoint x and w, and x not free in Φ:[3]

$$
\begin{aligned}
& wp(\llbracket\ \mathbf{var}\ x \bullet x, w : \left[P \middle/ R\right]\ \rrbracket, \Phi) \\
\equiv\ & (\forall x \bullet wp(x, w : \left[P \middle/ R\right], \Phi)) \\
\equiv\ & (\forall x \bullet P \wedge (\forall x, w \bullet R \Rightarrow \Phi)) \\
\equiv\ & (\forall x \bullet P) \wedge (\forall w \bullet (\exists x \bullet R) \Rightarrow \Phi) \\
\equiv\ & wp(w : \left[(\forall x \bullet P) \middle/ (\exists x \bullet R)\right], \Phi)
\end{aligned}
$$

Now, we have:

$$
\begin{aligned}
& \llbracket\ \mathbf{var}\ x \bullet x, w : \left[P_1 \middle/ R_1\right]\ \rrbracket \owedge \llbracket\ \mathbf{var}\ x \bullet x, w : \left[P_2 \middle/ R_2\right]\ \rrbracket \\
\equiv\ & w : \left[(\forall x \bullet P_1) \middle/ (\exists x \bullet R_1)\right] \owedge w : \left[(\forall x \bullet P_2) \middle/ (\exists x \bullet R_2)\right] \\
\equiv\ & w : \left[(\forall x \bullet P_1) \wedge (\forall x \bullet P_2) \middle/ (\exists x \bullet R_1) \wedge (\exists x \bullet R_2)\right] \\
\sqsubseteq\ & w : \left[(\forall x \bullet P_1 \wedge P_2) \middle/ (\exists x \bullet R_1 \wedge R_2)\right] \\
\equiv\ & \llbracket\ \mathbf{var}\ x \bullet w : \left[P_1 \wedge P_2 \middle/ R_1 \wedge R_2\right]\ \rrbracket
\end{aligned}
$$

□

3.4 A generalised distribution law

We can combine the above distribution laws, to obtain a law allowing us to distribute conjunction over two programs with some common structure involving sequence, alternation and variables declarations.

A *program schema* is like a program, except that it may contain occurrences of *program variables* (i.e. variables standing for programs). Semantically, a program schema is a function from (lists of) predicate transformers to predicate transformers.

We can capture the idea that programs S and T have some common structure by finding a program schema $\mathcal{F}$ and two lists of programs, $\hat{U} = (U_1, \cdots, U_n)$ and $\hat{V} = (V_1, \cdots, V_n)$, such that $S \equiv \mathcal{F}(\hat{U})$ and $T \equiv \mathcal{F}(\hat{V})$. The schema $\mathcal{F}$ embodies some common structure between S and T, while $\hat{U}$ and $\hat{V}$ are the components inserted into that structure to obtain S and T, respectively.

[3] If x occurs free in Φ, we must rename the local variable x, and all of its occurrences, to a fresh variable.

Note that in "matching" S and T to obtain $\mathcal{F}$, we can use properties such as commutativity of $[\!]$. We may also have to do some other rearrangements in order to obtain a match.

For example, consider

$$
\begin{array}{lcl}
S & \hat{=} & \textbf{if } A[i] < x \to lv := lv + 1 \;[\!]\; A[i] \geq x \to \textbf{skip fi} \\
T & \hat{=} & \textbf{if } A[i] > x \to hv := hv + 1 \;[\!]\; A[i] \leq x \to \textbf{skip fi}
\end{array}
$$

Splitting the $\geq$ and $\leq$ cases, we get

$$
\begin{array}{lcl}
S & \equiv & \mathcal{F}(lv := lv + 1, \textbf{skip}) \\
T & \equiv & \mathcal{F}(\textbf{skip}, hv := hv + 1)
\end{array}
$$

where

$$
\mathcal{F}(X, Y) \;\hat{=}\; \textbf{if } A[i] < x \to X \;[\!]\; A[i] = x \to \textbf{skip} \;[\!]\; A[i] > x \to Y \textbf{ fi}
$$

Lemma 1 If $S \equiv \mathcal{F}(\hat{U})$ and $T \equiv \mathcal{F}(\hat{V})$, for some schema $\mathcal{F}$, and lists of programs $\hat{U}$ and $\hat{V}$, then $S \owedge T \;\sqsubseteq\; \mathcal{F}(\hat{U} \owedge \hat{V})$, where $\hat{U} \owedge \hat{V} = (U_1 \owedge V_1, \cdots, U_n \owedge V_n)$.

Proof This follows by induction on the structure of $\mathcal{F}$.

If $\mathcal{F}$ contains no program constructors, then $\mathcal{F}$ is the identity schema, so $\hat{U} = (S)$ and $\hat{V} = (T)$. Thus, we have $S \owedge T \equiv \mathcal{F}(\hat{U}) \owedge \mathcal{F}(\hat{V}) \equiv \mathcal{F}(\hat{U} \owedge \hat{V})$.

If $\mathcal{F}$ is $\mathcal{F}_1; \mathcal{F}_2$, for some program schemas $\mathcal{F}_1$ and $\mathcal{F}_2$, then $S \equiv \mathcal{F}_1(\hat{U}'); \mathcal{F}_2(\hat{U}'')$ and $T \equiv \mathcal{F}_1(\hat{V}'); \mathcal{F}_2(\hat{V}'')$, where $\hat{U}' \frown \hat{U}'' = \hat{U}$ and $\hat{V}' \frown \hat{V}'' = \hat{V}$. Assuming the lemma holds for $\mathcal{F}_1$ and $\mathcal{F}_2$, we have:

$$
\begin{array}{ll}
S \owedge T & \\
\equiv & (\mathcal{F}_1(\hat{U}'); \mathcal{F}_2(\hat{U}'')) \;\owedge\; (\mathcal{F}_1(\hat{V}'); \mathcal{F}_2(\hat{V}'')) \\
\sqsubseteq & (\mathcal{F}_1(\hat{U}') \owedge \mathcal{F}_1(\hat{V}')) \;;\; (\mathcal{F}_2(\hat{U}'') \owedge \mathcal{F}_2(\hat{V}'')) \\
\sqsubseteq & \mathcal{F}_1(\hat{U}' \owedge \hat{V}') \;;\; \mathcal{F}_2(\hat{U}'' \owedge \hat{V}'') \\
\equiv & \mathcal{F}(\hat{U}' \owedge \hat{V}', \hat{U}'' \owedge \hat{V}'') \\
\equiv & \mathcal{F}(\hat{U} \owedge \hat{V})
\end{array}
$$

The case where $\mathcal{F}$ is an alternation follows in a similar manner, using the law for distributing conjunction over alternation.

$\square$

4 Recursion Blocks and Loops

We introduce recursion using recursion blocks, as in [Morgan, 1994]. A recursion block is written:

$$\textbf{re } X \bullet \mathcal{F}(X) \textbf{ er}$$

where X is a program variable, $\mathcal{F}(X)$ is a program schema possibly containing occurrences of X, and any occurrence of X in $\mathcal{F}(X)$ is to be treated as a recursive invocation of the recursion block.

In defining the semantics for a recursion block, we treat $\mathcal{F}$ as a function from predicate transformers to predicate transformers, and take the least fixed point of this function ([Morgan and Vickers, 1994]). Thus:

$$\textbf{re } X \bullet \mathcal{F}(X) \textbf{ er} \equiv \textbf{fix}(\mathcal{F})$$

where $\textbf{fix}(\mathcal{F})$ is the least fixed point of $\mathcal{F}$; i.e. the least refined program S such that $\mathcal{F}(S) \equiv S$. This least fixed point is the limit of a sequence of approximations defined by:

$$\begin{aligned} S_0 &\mathrel{\widehat{=}} \textbf{abort} \\ S_{i+1} &\mathrel{\widehat{=}} \mathcal{F}(S_i) \end{aligned}$$

Dijkstra's repetition construct is equivalent to a recursion block:

$$\textbf{do } G \rightarrow S \textbf{ od} \equiv \textbf{re } X \bullet \textbf{if } G \rightarrow S \,;\, X \;[\!]\; \neg G \rightarrow \textbf{skip fi er}$$

We can also express loops with multiple exit points (c.f. [King and Morgan, 1995]) as recursion blocks. For example, a loop:

$$\textbf{loop } S \,;\, \textbf{if } G \rightarrow T \,;\, \textbf{exit} \;[\!]\; \neg G \rightarrow U \textbf{ fi end}$$

can be written as:

$$\textbf{re } X \bullet S \,;\, \textbf{if } G \rightarrow T \;[\!]\; \neg G \rightarrow U \,;\, X \textbf{ fi er}$$

4.1 Conjoining Recursion Blocks

We now consider how to form the conjunction of two recursion blocks having some common structure.

Lemma 2 Suppose S and T are two recursion blocks with some common structure; i.e. for some program schema $\mathcal{F}$ and component statements $\hat{A}$ and $\hat{B}$, respectively, we have:

$$\begin{aligned} S &\equiv \textbf{re } X \bullet \mathcal{F}(X, \hat{A}) \textbf{ er} \\ T &\equiv \textbf{re } Y \bullet \mathcal{F}(Y, \hat{B}) \textbf{ er} \end{aligned}$$

where X and Y do not occur in $\hat{A}$ and $\hat{B}$. Then, $S \curlywedge T \sqsubseteq U$, where:

$$U \equiv \textbf{re } Z \bullet \mathcal{F}(Z, \hat{A} \curlywedge \hat{B}) \textbf{ er}$$

Proof Since S, T and U are all limits of sequences of approximations (say S_i, T_i and U_i), we prove that, for any i, $S_i \owedge T_i \sqsubseteq U_i$.

Base: $S_0 \owedge T_0 \equiv \textbf{abort} \owedge \textbf{abort} \equiv \textbf{abort} \equiv U_0$.

Step: Assuming $S_i \owedge T_i \sqsubseteq U_i$, we have:

$$
\begin{aligned}
& S_{i+1} \owedge T_{i+1} \\
\equiv\ & \mathcal{F}(S_i, \hat{A}) \owedge \mathcal{F}(T_i, \hat{B}) \\
\sqsubseteq\ & \mathcal{F}(S_i \owedge T_i, \hat{A} \owedge \hat{B}) \\
\sqsubseteq\ & \mathcal{F}(U_i, \hat{A} \owedge \hat{B}) \\
\equiv\ & U_{i+1}
\end{aligned}
$$

□

Example If we conjoin two programs, S and T, where:

$$
\begin{aligned}
S \ &\widehat{=}\ \textbf{re}\ X \bullet A\ ;\ \textbf{if}\ b_1 \to B \ [\!]\ b_2 \to C \ [\!]\ b_3 \to D\ ;\ X\ ;\ E\ \textbf{fi}\ \textbf{er} \\
T \ &\widehat{=}\ \textbf{re}\ Y \bullet A'\ ;\ \textbf{if}\ b_1 \to B' \ [\!]\ b_2 \to C' \ [\!]\ b_3 \to D'\ ;\ Y\ ;\ E'\ \textbf{fi}\ \textbf{er}
\end{aligned}
$$

We get:

$$
\begin{aligned}
S \owedge T \ \sqsubseteq\ & \textbf{re}\ Z \bullet \\
& \quad (A \owedge A')\ ; \\
& \quad \textbf{if}\ b_1 \to (B \owedge B') \\
& \quad [\!]\ b_2 \to (C \owedge C') \\
& \quad [\!]\ b_3 \to (D \owedge D')\ ;\ Z\ ;\ (E \owedge E') \\
& \quad \textbf{fi} \\
& \textbf{er}
\end{aligned}
$$

4.2 Repetition

The law for distributing conjunction over repetitions given in [Groves, 1998] is:

$\owedge$-*dist-rep:* $\textbf{do}\ G \to S'\ \textbf{od} \owedge \textbf{do}\ G \to T'\ \textbf{od} \sqsubseteq \textbf{do}\ G \to S' \owedge T'\ \textbf{od}$

This is obtained from the above law, by taking $\mathcal{F}(X, S) = \textbf{if}\ G \to S\ ;\ X\ [\!]\ \neg G \to \textbf{skip}\ \textbf{fi}$.

Example We can now conjoin the following programs (which count the number of values in $A[1..n]$ less that x and greater than x, respectively):

$$
\begin{array}{l}
[\![\ \textbf{var}\ i \bullet \\
\quad i, lv := 1, 0; \\
\quad \textbf{do}\ i \leq n \rightarrow \\
\qquad \textbf{if}\ A[i] < x \rightarrow lv := lv + 1 \\
\qquad [\!]\ A[i] \geq x \rightarrow \textbf{skip} \\
\qquad \textbf{fi}; \\
\qquad i := i{+}1 \\
\quad \textbf{od} \\
]\!]
\end{array}
\qquad\qquad
\begin{array}{l}
[\![\ \textbf{var}\ i \bullet \\
\quad i, hv := 1, 0; \\
\quad \textbf{do}\ i \leq n \rightarrow \\
\qquad \textbf{if}\ A[i] > x \rightarrow hv := hv + 1 \\
\qquad [\!]\ A[i] \leq x \rightarrow \textbf{skip} \\
\qquad \textbf{fi}; \\
\qquad i := i{+}1 \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

to obtain (after a little simplification):

$$
\begin{array}{l}
[\![\ \textbf{var}\ i \bullet \\
\quad i, lv, hv := 1, 0, 0; \\
\quad \textbf{do}\ i \leq n \rightarrow \\
\qquad \textbf{if}\ A[i] < x \rightarrow lv := lv + 1 \\
\qquad [\!]\ A[i] = x \rightarrow \textbf{skip} \\
\qquad [\!]\ A[i] > x \rightarrow hv := hv + 1 \\
\qquad \textbf{fi}; \\
\qquad i := i{+}1 \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

5 Conclusions

We have shown how a conjunction of recursion blocks with similar structure can be merged to obtain a single recursion block. The properties of the program conjunction operator make this very straightforward: if there is any inconsistency, you end up with magic, not with an incorrect program. It may be worth checking feasibility along the way, to avoid going too far down the wrong path — though in our experience, an informal check is usually sufficient.

Basing the technique on recursion blocks, rather than loops, makes the reasoning much simpler, as well as allowing a more general law to be obtained. The main reason for this is that with a recursion block, all cases are treated explicitly, including the exit case, which is implicit in a **do-od** loop. The fact that different exit conditions can be handled separately, rather than being combined into a single condition is also helpful.

Of course, this process is still not completely mechanical. If the programs being conjoined do not already have the same structure, it may take some ingenuity to get them into a form where they can be combined (and in some cases it would be preferable to implement them independently anyway). The main premiss underlying this work is that it is easier to make such changes on the derivation of a program that on the final code.

The major shortcoming of the technique presented here is that it does not handle loops and recursion blocks with different guards. For example, we would like to be able to conjoin the following programs (which find the first occurrence of x, respectively y, in $A[1..n]$, where such occurrences are guaranteed to exist):

$$\begin{array}{l} i := 1; \\ \mathbf{do}\ A[i] \neq x \rightarrow i := i + 1\ \mathbf{od} \end{array} \qquad\qquad \begin{array}{l} j := 1; \\ \mathbf{do}\ A[j] \neq y \rightarrow j := j + 1\ \mathbf{od} \end{array}$$

to obtain:

$$\begin{array}{l}
i, j := 1, 1; \\
\mathbf{loop} \\
\quad \begin{array}{lll}
\mathbf{if}\ A[i] \neq x \ \wedge\ A[j] \neq y & \rightarrow & i, j := i+1, j+1 \\
[\!]\ A[i] \neq x \ \wedge\ A[j] = y & \rightarrow & i := i + 1; \\
 & & \mathbf{do}\ A[i] \neq x \rightarrow i := i + 1\ \mathbf{od}; \\
 & & \mathbf{exit} \\
[\!]\ A[i] = x \ \wedge\ A[j] \neq y & \rightarrow & j := j + 1; \\
 & & \mathbf{do}\ A[j] \neq y \rightarrow j := j + 1\ \mathbf{od}; \\
 & & \mathbf{exit} \\
[\!]\ A[i] = x \ \wedge\ A[j] = y & \rightarrow & \mathbf{exit} \\
\mathbf{fi} & &
\end{array} \\
\mathbf{end}
\end{array}$$

This combination requires a more general form of laws for alternations (which is given in [Groves, 1998]), a more fine grained approach to identifying common structure (allowing guards, rather than just statements, to be abstracted), and a more complex approach to merging recursion blocks. The details will be reported elsewhere.

Acknowledgements

Thanks to Ray Nickson for comments and corrections on earlier drafts of this paper.

References

[Ainsworth and Wallis, 94] Mike Ainsworth and Peter Wallis. "Co-Refinement". *Proc. 6th Refinement Workshop*, David Till (Ed), Springer-Verlag, 1994, pp151–166.

[Back and Butler, 1996] R. J. R Back and M. J. Butler. "Fusion and Simultaneous Execution in the Refinement Calculus". *Acta Informatica*, **35**, 11 (1998), pp921–949.

[Groves, 1998] Lindsay Groves. "Adapting Program Derivations using Program Conjunction". International Refinement Workshop and Formal Methods Pacific 1998 (IRW/FMP'98), Canberra, Springer-Verlag.

[Hehner, 1993] Eric C. R. Hehner. *A Practical Theory of Programming.* Springer-Verlag, 1993.

[Hoare *et al*, 1987] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey and B. A. Sufrin "Laws of Programming". *Communications of the ACM*, **30** (1987), pp672–686.

[King and Morgan, 1995] S. King and C. C. Morgan. "Exits in the refinement calculus". *Formal Aspects of Computing*, **7**, 1 (1995), pp54–76.

[Mahony, 1998] Brendan Mahony. "The Least Conjunctive Refinement and Promotion in the Refinement Calculus". Draft paper, 1998.

[Morgan, 1994] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, Second Edition, 1994.

[Morgan and Vickers, 1994] C. C. Morgan and T. N. Vickers. *On the Refinement Calculus.* FACIT Series in Computer Science. Springer Verlag, 1994.

[Utting, 92] Mark Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning.* PhD thesis, University of New South Wales, 1992.

[Ward, 93] Nigel Ward. "Adding specification constructors to the refinement calculus". *FME'93: Industrial-Strength Formal Methods*, J. C. P. Woodcock and P. G. Larsen (Eds), Springer-Verlag, 1993, pp652–670.