# New Programs From Old

G. RAMALINGAM and THOMAS REPS

University of Wisconsin−Madison

This paper studies operations for creating new variants of a program that relate, in a well-defined way, to existing variants of the program. We formalize a *program modification* as a (special kind of) function from programs to programs, and study the algebra of these program modifications. We make use of the algebraic structure to formalize several intuitive concepts, such as that of "compatibility among program modifications", and establish several new results concerning the problems of *program merging* and *separating consecutive edits*. We also identify a category in which the objects are programs and the morphisms are program modifications, and show how program integration relates to the pushout in this category.

## 1. Introduction

In the program-development process, there is often the need to create new variants of a program that relate, in some specific way, to existing variants of the program. Examples of operations to create new program variants include

*Merging software extensions*

> Given two programs *a* and *b*, the goal is to generate a program that "has all the capabilities of each of the original programs", that is, a program that "combines the features of both *a* and *b*".

*Program integration*

> Given *three* versions of a program—an original *base* version and two independent revisions *a* and *b* of *base*, the goal is to determine if the two changes made to *a* and *b* have

This work was supported in part by an IBM Graduate Fellowship, by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602 and CCR-9100424, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by a grant from the Digital Equipment Corporation.

A preliminary version of this work appeared in the *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology (AMAST)*, (Iowa City, Iowa, May 22-25, 1991) [10].

Authors' address: Computer Sciences Department, University of Wisconsin−Madison, 1210 W. Dayton St., Madison, WI 53706.
E-mail: {ramali, reps}@cs.wisc.edu

an undesirable semantic interaction, and if they do not, create a program that incorporates the two changes (as well as what is common to all three programs).

*Separating consecutive edits*

Consider the case of two consecutive edits to *base*; let *a* be a program obtained by modifying *base*, and let *c* be a program obtained by modifying *a*. By "separating consecutive edits" we mean creating a program that includes the second modification but not the first.

*Propagating changes through multiple versions*

Suppose a tree or dag of related versions of a program exists. The goal is to make the same enhancement or bug-fix to all of them.

Several other papers[2, 6, 15] have investigated such operations at the concrete level (*i.e.*, by defining algorithms for them that work on particular concrete representations of programs). In this paper, we investigate such operations at an abstract level. We make use of an *algebra* that abstracts away from the details of particular (concrete) operations for the problems described above. This approach permits us to study the various program-manipulation problems by investigating their algebraic properties. (A simple example of an algebraic property—which a specific program-integration operation might or might not possess—is that of associativity. In this context associativity means: "If three variants of a given base are to be integrated by a pair of two-variant integrations, the same result is produced no matter which two variants are integrated first.")

This paper is a follow-on to an earlier study of the algebraic properties of program integration by Reps [14]. However, whereas Reps's work concerns a homogeneous algebra of *programs*, the work reported in this paper concerns a heterogeneous algebra of *programs* and *program modifications*. In this paper, we formalize the concept of a program modification—the change made to one program to obtain another—as a (special kind of) function from programs to programs. We study an algebra of such program modifications and apply our results about the algebraic properties of such an algebra to questions about the program-manipulation operations listed above.

While this paper focuses primarily on the algebraic structure of the system under consideration, we wish to point out that the concepts and operations discussed in the paper *can* all be implemented as concrete operations on the kinds of program representations used in previous work on concrete algorithms for program integration.

The remainder of the paper is organized into seven sections. In Section 2, we review previous work in which program integration was formulated as an operation in a Brouwerian algebra. In Section 3, we formalize a program modification as a special kind of function from programs to programs and define a category in which the objects denote programs and the morphisms denote program modifications. (Though we will use commutative diagrams of this category to succinctly represent many algebraic properties of program modifications in the earlier sections, knowledge of category theory is essential only in Section 8.) In Section 4, we define a partial ordering on program modifications, study how it relates to function composition, and show that the partial ordering naturally captures the notion of subsumption among modifications. We also formalize the notion of *compatibility* among modifications. In Section 5, we define and study further operations for combining program modifications (which, among other things, can be used define a program integration operator). In Section 6, we utilize results from previous sections to establish properties of the integration operator. In Section 7, we study several alternative solutions to the problem of separating consecutive edits. In Section 8, we consider the question of how program integration relates to the pushout in the category defined in Section 3.

## 2. Previous Work: An Algebra Of Programs

In this section, we briefly review the Brouwerian algebraic framework for program integration developed in [14]. The idea behind this approach may be informally described as follows. A program consists of one or more program components. Hence, a program may be viewed as a set of program components. However, not every set or collection of program components is a program. The integration algorithm described by Horwitz *et al.* [6], referred to as the HPR algorithm hereafter, decomposes a program into its constituent components, compares two programs *a* and *base* and determines what components were added to *base* and what components of *base* were preserved in developing program *a* from *base*. Let $a \dot{-} base$ denote the set of program components in *a* but not in *base*. (The $\dot{-}$ operator is related to, but not the same as, the set difference operator, as will be explained shortly.) Let $a \sqcap base$ denote the set of program components that are in both *a* and *base*. The integration algorithm integrates variants *a* and *b* of *base* by putting together the program components in $a \dot{-} base$,

$b \dot{-} base$ and $a \sqcap base \sqcap b$, obtaining $(a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base)$.

**Definition 2.1.** [8] A *Brouwerian algebra* is an algebra $(P, \sqcup, \sqcap, \dot{-}, \top)$ where

(i) $(P, \sqcup, \sqcap)$ is a lattice (with $\sqcup$ denoting the join operator, and $\sqcap$ denoting the meet operator) with greatest element $\top$. The corresponding partial order will be denoted by $\sqsubseteq$.

(ii) For all $a$, $b$, and $c$ in $P$, $(a \dot{-} b) \sqsubseteq c$ iff $a \sqsubseteq (b \sqcup c)$.

It can be shown that a Brouwerian algebra has a least element, given by $\top \dot{-} \top$, which will be denoted by $\perp$.

**Definition 2.2.** [8] A *double Brouwerian algebra* is an algebra $(P, \sqcup, \sqcap, \dot{-}, \div, \top)$ where both $(P, \sqcup, \sqcap, \dot{-}, \top)$ and $(P, \sqcap, \sqcup, \div, \top \dot{-} \top)$ are Brouwerian algebras. In particular,

(i) For all $a$, $b$, and $c$ in $P$, $(a \div b) \sqsupseteq c$ iff $a \sqsupseteq (b \sqcap c)$.

The operator $\dot{-}$ of a double Brouwerian algebra will be referred to as the pseudo-difference operator, while the operator $\div$ will be referred to as the quotient operator.

**Definition 2.3.** [14] A ternary operator _[_]_, called the *integration* operator, of a Brouwerian algebra is defined as follows:

$$a\,[base\,]\,b \;=_{def}\; (a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base).$$

We now describe a concrete example of a Brouwerian algebra of programs. The HPR integration algorithm makes use of a program representation called a *program dependence graph* [3, 7]. Let $s$ be a vertex of a program dependence graph $G$. The *slice* of $G$ with respect to $s$, denoted by $G/s$, is a graph induced by all vertices on which $s$ has a transitive flow or control dependence. A dependence graph $G$ is said to be a *single-point slice* if it is the slice of some dependence graph with respect to some vertex (*i.e.*, equivalently, iff $G = G/v$ for some vertex $v$ in $G$). Let $G_1$ denote the set of all single-point slices. A partial order $\sqsubseteq$ is defined on the set $G_1$ as follows: if $a$ and $b$ are single-point slices, $b \sqsubseteq a$ iff $b$ is a slice of $a$ with respect to some vertex in $a$ (*i.e.*, iff $b = a/v$ for some vertex $v$ in $a$). Thus, $\sqsubseteq$ denotes the relation “is-a-subslice-of”.

The HPR integration algorithm treats the single-point slices of a program (dependence graph) as the program’s indivisible components in that if one of the variants contains a single-point slice $s$ not in the base program, then the integrated program will contain the slice $s$ in its entirety. The integrated program is essentially constructed from the single-point slices occurring in the base program and the variants. The semantic justification of the algorithm follows from the results of [12], which relates the semantics of a slice of a program to the semantics of the program itself.

A subset $A$ of $G_1$ is said to be a *downwards-closed set* if for every $x \in A$ and $y \sqsubseteq x$, $y \in A$. Given a subset $A$ of $G_1$, $DC(A)$, the downwards-closed set generated by $A$, is defined as

$$DC(A) =_{def} \{ b \in G_1 \mid \exists a \in A. (b \sqsubseteq a) \}.$$

For any program $p$ let $S_p$ denote the set of all single-point slices occurring in $p$ (that is, in the program dependence graph of $p$). The set $S_p$ is obviously a downwards-closed set. Thus, if $A$ is an arbitrary set of single-point slices, then $DC(A)$ represents the smallest program that contains all the slices in $A$. It can be verified that $DC(A \cup B) = DC(A) \cup DC(B)$, and that a set $A$ is downwards-closed iff $DC(A) = A$. The concepts of an *upwards-closed set*, and $UC(A)$, the upwards-closed set generated by a set A, are similarly defined.

$$UC(A) =_{def} \{ b \in G_1 \mid \exists a \in A. (a \sqsubseteq b) \}.$$

Let $DCS$ denote the set of all downwards-closed sets of single-point slices. Let $\top$ denote the set $G_1$, and let $\perp$ denote the empty set. Let $\cup$, $\cap$, and $-$ denote the set-theoretic union, intersection and difference operators. $DCS$ is closed with respect to $\cup$ and $\cap$. $DCS$ is not closed under $-$, but is closed under the pseudo-difference operator $\dot{-}$ defined as follows:

$$X \dot{-} Y = DC(X - Y)$$

$DCS$ is also closed under a similar (dual) operator $\div$ defined as follows, where $\overline{\phantom{x}}$ denotes the set-theoretic complement with respect to $\top$:

$$X \div Y = \overline{UC(Y - X)}.$$

**Theorem 2.4.** (Reps[14]) $(DCS, \cup, \cap, \dot{-}, \div, \top)$ is a double Brouwerian algebra.

Reps also shows that the HPR algorithm corresponds to an implementation of the _[_]_ operator of this double Brouwerian algebra and uses the properties of Brouwerian algebra to prove various algebraic properties of program-integration.

**Example 2.5.** The following example illustrates the HPR algorithm. Each program in this figure contains an *end statement* that names a list of variables used in the program whose final values are *returned* by the execution of the program. In describing the Brouwerian algebraic framework, the programs are used in the figure as a representation for a set of single-point slices, namely the downwards closure of the set of slices of the respective programs with respect to the final value computed for each of the variables listed in the end statement.

| $a$ | $base$ | $b$ | $a \dot{-} base$ | $a \cap base \cap b$ | $b \dot{-} base$ | $a\,[base\,]b$ |
|---|---|---|---|---|---|---|
| **program**<br>$x := 0$<br>**end**$(x)$ | **program**<br>$x := 0;$<br>$y := x$<br>**end**$(x, y)$ | **program**<br>$x := 0;$<br>$y := x;$<br>$z := y$<br>**end**$(x, y, z)$ | $\varnothing$ | **program**<br>$x := 0$<br>**end**$(x)$ | **program**<br>$x := 0;$<br>$y := x;$<br>$z := y$<br>**end**$(z)$ | **program**<br>$x := 0;$<br>$y := x;$<br>$z := y$<br>**end**$(x, z)$ |

This example illustrates one subtlety of the HPR algorithm: an insertion made in one integrand can "override" a deletion in the other integrand. In the example given above, the insertion of statement $z := y$ in integrand $b$ overrides the deletion of $y := x$ from integrand $a$ because $z := y$ uses the value assigned to $y$ by $y := x$. Note that the deletion from $a$ of the final use of $y$ does *not* get overridden. □

The appendix contains a list of the more important laws that hold in double Brouwerian algebras. The reader is referred to [8, 11, 13, 14] for a more comprehensive discussion of the algebraic laws that hold in Brouwerian algebras and double Brouwerian algebras. It is worth remembering that every Brouwerian algebra is a distributive lattice, and that every Boolean algebra is a double Brouwerian algebra (with the pseudo-difference and quotient operators being defined appropriately).

The rest of the paper assumes only that the set of program (representations) have a double Brouwerian algebraic structure. Consequently, the results obtained are not restricted to the above model based on the program dependence graph representation, but apply to any double Brouwerian algebraic structure whatsoever. Thus, one could potentially define similar double Brouwerian algebras based on other representations of programs to which the results of this paper would carry over.

## 3. Towards An Algebra Of Program Modifications

### 3.1. The fm-algebraic framework

The goal of program-integration is to merge or combine changes made to some program. The concept of a "change made to a program" or a *program-modification* was formalized in [9] and made use of in providing an alternative definition of integration. We briefly review this formalism below.

**Definition 3.1.** A *modification algebra* is a heterogeneous algebra (P,M,$\Delta$, +, *apply*) with two sets P and M and three operations $\Delta$, +, and *apply*, with the following functionalities:

$$\Delta:\ P \times P \longrightarrow M$$
$$+:\ M \times M \longrightarrow M$$
$$apply:\ M \times P \longrightarrow P.$$

For our purposes, we may interpret the components of a modification algebra as follows. P denotes the set of programs; M denotes a set of allowable program-modification operations; $\Delta(a, base)$ yields the program modification performed on *base* to obtain *a*[1]; operation $m_1 + m_2$ combines two modifications $m_1$ and $m_2$ to give a new modification; $apply(m, base)$ denotes the program obtained by performing modification *m* on program *base*.

**Definition 3.2.** The 2-variant integration operator $\_[\![\_]\!]\_$: $P \times P \times P \longrightarrow P$ of a modification algebra is defined as follows:

$$a[\![base]\!]b =_{def} apply(\Delta(a, base) + \Delta(b, base),\ base).$$

The UNIX[2] utility *diff* yields a simple example of a modification algebra. Here, elements of *P* are text-files and elements of *M* are script files (for an editor). The program *diff* (with the option −*e*) implements $\Delta$, while the editor *ed* implements *apply*. The integration operation that results if we define the operator + to be file concatenation is essentially the UNIX utility *diff3*. However, other definitions of the operator + are possible too. Fraser [4], for instance, outlines a method for merging two editor command scripts into a single “equivalent” one that leads to a different integration algorithm. These text-based integration algorithms, however, differ from the HPR integration algorithm in that they do not provide any guarantees about how the semantics of the integrated program relates to the semantics of the programs that are arguments to the integration operation.

---

[1] Of course, there need not be a unique program-modification in M that yields *a* when applied to *base*. A more precise interpretation of $\Delta(a, base)$, in the algebra we are interested in, will be given in Section 5.1.

[2] UNIX is a trademark of AT&T Bell Laboratories.

We now consider a particular kind of modification algebra:

**Definition 3.3.** A *functional-modification algebra* (abbreviated *fm*-algebra) is a modification algebra (P,M,$\Delta$, + ,*apply*) in which M $\subseteq$ P $\longrightarrow$ P and *apply* is ordinary function application.

The HPR integration algorithm can be modelled as the integration operation of an appropriately defined *fm*-algebra. The definition of this *fm*-algebra and the development of its algebraic structure are the subject of the following sections. In section 3.2 we define the set of programs and the set of program-modifications of interest. In sections 4 and 5 we study the algebraic structure of the set of program-modifications and define the *fm*-algebra operators. In section 6 we study the integration operation.

### 3.2. Modifications and Their Representations

In the remaining part of the paper, let $\mathcal{P} = (P, \sqcup, \sqcap, \dot{-}, \div, \top)$ denote a given double Brouwerian algebra. The elements of $P$ will be referred to as *programs*. The letters $x$, $y$, $z$, $a$, $b$, $c$, and *base*, possibly subscripted, will typically denote elements of $P$. The set $M$ of program-modifications of interest is given by the following definition.

**Definition 3.4.** $M$, a subset of $P \longrightarrow P$, is defined as follows:

$$M =_{def} \{ <x,y> \;\mid\; x,y \in P, x \sqsubseteq y, \text{ where } <x,y> =_{def} \lambda z.\ (z \sqcap y) \sqcup x \}$$

The elements of $M$ will be referred to as *modifications*. The symbol $m$, possibly subscripted, will typically denote an element of $M$.

Note that $<x,y>$ denotes a function from $P$ to $P$. Informally, we may think of modification $\lambda z.\ (z \sqcap y) \sqcup x = (\lambda z.\ z \sqcup x) \circ (\lambda z.\ z \sqcap y)$[3] as consisting of two components: $\lambda z.\ z \sqcup x$, which represents an addition of new program components (where $x$ is the set of program components to be added), and $\lambda z.\ z \sqcap y$, which represents deletion of certain program components (where $y$ represents the set of program components to be preserved).

Why do we restrict attention to modifications $<x,y>$ satisfying $x \sqsubseteq y$? Observe that any function $\lambda z.\ (z \sqcap y_1) \sqcup x_1 = \lambda z.\ (z \sqcap (y_1 \sqcup x_1)) \sqcup x_1 = <x_1, y_1 \sqcup x_1>$. Consequently, $M$ does include *all* functions of the form $\lambda z.\ (z \sqcap y) \sqcup x$. However, when we use the notation $<x,y>$,

---

[3]Here $\circ$ denotes function composition. In particular, $m_1 \circ m_2$ denotes the function $\lambda x.\ m_1(m_2(x))$.

we restrict attention to $x$ and $y$ satisfying $x \sqsubseteq y$ so that the *representation* of a modification is unique, as the following proposition shows.

**Proposition 3.5.** $<x_1, y_1> = <x_2, y_2>$ iff $x_1 = x_2$ and $y_1 = y_2$ (given $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$).

**Proof.** Consider $m = <x,y> = \lambda z.\ (z \sqcap y) \sqcup x$, where $x \sqsubseteq y$. Note that $m(\bot) = (\bot \sqcap y) \sqcup x = x$, and that $m(\top) = (\top \sqcap y) \sqcup x = y \sqcup x = y$. It follows that $m$ uniquely identifies $x$ and $y$. The result follows. □

In summary, we are interested in the set of functions $M \subseteq P \longrightarrow P$. Each function in $M$ has a unique representation as an ordered pair of elements in $P$. We denote the set of all representations of the modifications by $S$:

**Definition 3.6.** $S$, a subset of $P \times P$, is defined as follows:

$$S =_{def} \{ (x,y) \in P \times P \mid x \sqsubseteq y \}.$$

Figure 3.1 presents some examples of programs, modifications, and the application of modifications to programs, when the underlying double Brouwerian algebra $\mathcal{P}$ is the algebra $(DCS, \cup, \cap, \dot{-}, \div, \top)$ discussed in Section 2. In this figure, the "programs" with the **end** statements are used as a representation for the set of the slices of the respective programs with respect to the final value computed for each of the variables listed in the end statement. Entries in the first table denote programs while entries in the second table denote modifications. The first modification, titled[4] $\Delta(a, base)$, represents the addition of the computation of (the final value of) $p$. The second modification, titled $\Delta(b, base)$, denotes the simultaneous addition of the computation of $r$ and the deletion of the computation of $s$. Similarly the third modification, titled $\Delta(a, base) + \Delta(b, base)$, denotes the simultaneous addition of the computation of $r$ and $p$ and the deletion of the computation of $s$. The third table illustrates a program obtained by applying a modification to another program.

**Proposition 3.7.**

($a$) $M$ is closed with respect to $\circ$. In particular,

$$<x_1, y_1> \circ <x_2, y_2> = <x_1 \sqcup (x_2 \sqcap y_1), y_1 \sqcap (y_2 \sqcup x_1)>.$$

---

[4]The reader may ignore these titles for now. These fm-algebraic operations will be defined later on.

| $base$ | $a$ | $b$ | $a[\![base]\!]b$ |
|---|---|---|---|
| $DC\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ s := x+y\\ \textbf{end}(s)\end{array}\right]$ | $DC\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ s := x+y\\ \ \ p := x*y\\ \textbf{end}(s,p)\end{array}\right]$ | $DC\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ r := x/y\\ \textbf{end}(r)\end{array}\right]$ | $DC\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ r := x/y\\ \ \ p := x*y\\ \textbf{end}(r,p)\end{array}\right]$ |

| $\Delta(a,base)$ | $\Delta(b,base)$ | $\Delta(a,base) + \Delta(b,base)$ |
|---|---|---|
| $<DC\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ p := x*y\\ \textbf{end}(p)\end{array}\right],\top>$ | $<DC\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ r := x/y\\ \textbf{end}(r)\end{array}\right],\overline{UC}\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ s := x+y\\ \textbf{end}(s)\end{array}\right]>$ | $<DC\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ r := x/y\\ \ \ p := x*y\\ \textbf{end}(r,p)\end{array}\right],\overline{UC}\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ s := x+y\\ \textbf{end}(s)\end{array}\right]>$ |

| $c$ | $\Delta(b,base)(c)$ |
|---|---|
| $DC\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ q := y/x\\ \textbf{end}(q)\end{array}\right]$ | $DC\left[\begin{array}{l}\textbf{program}\\ \ \ x := 4\\ \ \ y := 2\\ \ \ q := y/x\\ \ \ r := x/y\\ \textbf{end}(q,r)\end{array}\right]$ |

**Figure 3.1.** An illustration of various values in the *fm*-algebra.

($b$) Let $I$ denote the modification $<\perp,\top>$. Then, $(M, \circ, I)$ is a monoid: *i.e.*, $\circ$ is associative, and $I$ is the identity with respect to $\circ$.

**Proof.**

$$(a)\ <x_1,y_1> \circ <x_2,y_2> \ = \ (\lambda z.\ (z \sqcap y_1) \sqcup x_1) \ \circ \ (\lambda z.\ (z \sqcap y_2) \sqcup x_2)$$

$$= \lambda z.\ (\ ((z \sqcap y_2) \sqcup x_2) \ \sqcap y_1) \sqcup x_1$$

$$= \lambda z.\ ((z \sqcap y_2 \sqcap y_1) \sqcup (x_2 \sqcap y_1)) \sqcup x_1$$

(since a Brouwerian algebra is a distributive lattice)

$$= \lambda z.\ (z \sqcap (y_2 \sqcap y_1)) \sqcup ((x_2 \sqcap y_1) \sqcup x_1)$$

We observed earlier that M includes every function of the form $\lambda z.\ (z \sqcap y) \sqcup x$. Hence, $<x_1,y_1> \circ <x_2,y_2> \in M$. As observed in the proof of Proposition 3.5, the unique representation of any $m \in M$ is $<m(\perp), m(\top)>$. Let $m$ denote the above function $\lambda z.\ (z \sqcap (y_2 \sqcap y_1)) \sqcup ((x_2 \sqcap y_1) \sqcup x_1)$. It can be verified easily that $m(\perp)$ is $x_1 \sqcup (x_2 \sqcap y_1)$ and that $m(\top)$ is $y_1 \sqcap (y_2 \sqcup x_1)$. Hence,

$<x_1,y_1> \circ <x_2,y_2> = <x_1 \sqcup (x_2 \sqcap y_1), y_1 \sqcap (y_2 \sqcup x_1)>$.

($b$) Function composition is associative. The modification $<\bot,\top>$ denotes the function $\lambda z.\ (z \sqcap \top) \sqcup \bot = \lambda z.z$ which is the identity function. It follows that $(M, \circ, I)$ is a monoid. □

In lectures about the program-integration problem, we have often used pictures like the one shown in Figure 3.2 to provide an intuitive explanation of the goals of the work. On numerous occasions, audience members have conjectured that integration was a pushout in an appropriate category. Pictures depicting integration typically have programs and arrows between programs, representing modifications to programs. We may formalize these pictures as diagrams in a category we define below, a category in which the objects denote programs, and arrows denote program-modifications. We will return to the question of whether integration is a pushout in Section 8. (Knowledge of category theory is not essential in the earlier sections and the material relating to category theory appears indented in the earlier sections to make it easier to skip such material.)

**Definition 3.8.** Define a category $\mathcal{C}$ as follows: The set of objects is $P$. For every $a \in P$, and $m \in M$, there is a morphism labelled $[a, m, m(a)]$ with domain $a$ and codomain $m(a)$. Thus, every morphism $[a, m, m(a)]$ is associated with a program modification $m$. (If no confusion is likely, the morphism will just be denoted by $m$.) The operation $\circ$ is defined to be function composition. Thus, $[b, m_2, c] \circ [a, m_1, b] = [a, m_2 \circ m_1, c]$.

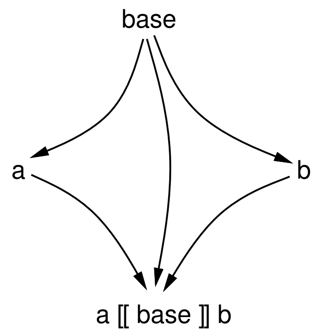


**Figure 3.2.** The problem of two-variant program integration.

Note that multiple morphisms can be associated with the same modification function, provided each of these morphisms has a different domain. Every set of morphisms has an associated set of modifications. Occasionally, we blur the distinction between a set of morphisms and the associated set of modifications, in order to give a simple pictorial interpretation for certain sets of modifications.

## 4. A Subsumption Ordering Among Program Modifications

In this section we define a partial ordering $\leq$ on the set $M$ of modifications (introduced in the previous section) that captures the notion of subsumption among modifications. In particular, we may think of "$m_1 \leq m_2$" as a synonym for "modification $m_1$ is *part of* modification $m_2$". We will show that any two modifications $m_1$ and $m_2$ have a greatest lower bound $m_1 \sqcap m_2$ (which we may think of as the modification that is "common" to the two modifications $m_1$ and $m_2$). We will see that not every pair of modifications has a least upper bound. We will, in fact, see that $M$ partially inherits a Brouwerian algebraic structure.

### 4.1. The Partial Ordering

**Definition 4.1.** Define a binary relation $\leq$ on $M$ as follows:

$$<x_1,y_1> \ \leq \ <x_2,y_2> \ \Leftrightarrow \ (x_1 \sqsubseteq x_2) \text{ and } (y_1 \sqsupseteq y_2)$$

This is obviously a partial ordering on modifications. Note that the ordering on the respective second components is the dual of the ordering on the respective first components. We will see later why this partial ordering is interesting. To understand the structure of this partial order better, consider the algebraic structure of the set $P \times P$ with respect to the partial ordering on its elements defined by $(x_1,y_1) \sqsubseteq' (x_2,y_2)$ iff $x_1 \sqsubseteq x_2$ and $y_1 \sqsupseteq y_2$. This is trivially a double Brouwerian algebra, namely the product algebra of the original double Brouwerian algebra $\boldsymbol{\mathcal{P}}$ and its dual $\overline{\boldsymbol{\mathcal{P}}}$. Let $\sqcup'$, $\sqcap'$, $\dot{-}'$, $\div'$ denote the corresponding Brouwerian algebraic operators on $\boldsymbol{\mathcal{P}} \times \overline{\boldsymbol{\mathcal{P}}}$.

Now consider the set $S \subseteq P \times P$ (of representations of modifications). Observe that $(M, \leq)$ is isomorphic to $(S, \sqsubseteq')$. Let us denote the restriction of $\sqcup'$, $\sqcap'$, $\dot{-}'$, and $\div'$ to $M$ (using the isomorphism mentioned above) by $\sqcup$, $\sqcap$, $\dot{-}$, and $\div$ respectively. It turns out that $\sqcap$ and $\dot{-}$ are total operations on M, while $\sqcup$ and $\div$ are partial operations on $M$.

**Proposition 4.2.**

($a$) $<x_1,y_1> \sqcap <x_2,y_2> = <x_1 \sqcap x_2, y_1 \sqcup y_2>$

($b$) $<x_1,y_1>$ and $<x_2,y_2>$ have a least upper bound iff $x_1 \sqsubseteq y_2$ and $x_2 \sqsubseteq y_1$, in which case the least upper bound is $<x_1 \sqcup x_2, y_1 \sqcap y_2>$.

($c$) $<x_1,y_1> \dot{-} <x_2,y_2> = <x_1 \dot{-} x_2, y_1 \div y_2>$

**Proof.**

($a$) It follows from Definition 4.1 that the meet of $<x_1,y_1>$ and $<x_2,y_2>$ must be $<x_1 \sqcap x_2, y_1 \sqcup y_2>$, provided $<x_1 \sqcap x_2, y_1 \sqcup y_2> \in M$. Since $<x_1,y_1> \in M$, we know that $x_1 \sqsubseteq y_1$. Hence $x_1 \sqcap x_2 \sqsubseteq y_1 \sqcup y_2$ and $<x_1 \sqcap x_2, y_1 \sqcup y_2> \in M$.

($b$) It follows from Definition 4.1 that the join of $<x_1,y_1>$ and $<x_2,y_2>$ must be $<x_1 \sqcup x_2, y_1 \sqcap y_2>$, provided $<x_1 \sqcup x_2, y_1 \sqcap y_2> \in M$. Now $<x_1 \sqcup x_2, y_1 \sqcap y_2> \in M$ iff $x_1 \sqcup x_2 \sqsubseteq y_1 \sqcap y_2$. Since we already have $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$, $x_1 \sqcup x_2 \sqsubseteq y_1 \sqcap y_2$ iff $x_1 \sqsubseteq y_2$ and $x_2 \sqsubseteq y_1$.

($c$) This follows just as in ($a$). $\square$

We know that the partial algebra $(M, \sqcup, \sqcap, \dot{-}, \div)$ can be *extended* to be a complete double Brouwerian algebra (namely $\mathcal{P} \times \overline{\mathcal{P}}$). What this says is that $M$ inherits many of the algebraic properties of double Brouwerian algebra. In particular, any identity that holds in a double Brouwerian algebra holds in $M$ too, as long as all the relevant expressions are defined (i.e., as long as they exist). The following are some useful properties which are a consequence of this fact.

**Proposition 4.3.**

($a$) $M$ is *consistently complete*: any two modifications $m_1$ and $m_2$ have a least upper bound iff they have an upper bound.

($b$) $m_1 = (m_1 \dot{-} m_2) \sqcup (m_1 \sqcap m_2)$

($c$) $(m_1 \sqcup m_2) \sqcap m_3 = (m_1 \sqcap m_3) \sqcup (m_2 \sqcap m_3)$, provided $(m_1 \sqcup m_2) \sqcap m_3$ exists.

($d$) $(m_1 \dot{-} m_2) \dot{-} m_2 = m_1 \dot{-} m_2$

($e$) If $(m_1 \sqcup m_2) \geq m_3$ then $m_2 \geq (m_3 \dot{-} m_1)$.

**Proof.**

($a$) If $<x_1,y_1>$ and $<x_2,y_2>$ have an upper bound $<x,y>$, then $x_1 \sqcup x_2 \sqsubseteq x \sqsubseteq y \sqsubseteq y_1 \sqcap y_2$. Hence, it follows from Proposition 4.2($b$) that $<x_1,y_1>$ and $<x_2,y_2>$ have a least upper bound.

($b$) Note that $(m_1 \dot{-} m_2)$ and $(m_1 \sqcap m_2)$ have an upper bound, namely $m_1$. Consequently, from ($a$), $(m_1 \dot{-} m_2) \sqcup (m_1 \sqcap m_2)$ exists. Hence, the equality $m_1 = (m_1 \dot{-} m_2) \sqcup (m_1 \sqcap m_2)$ follows directly from the same equality of Brouwerian algebra.

($c$) $m_3$ is an upper bound for both $(m_1 \sqcap m_3)$ and $(m_2 \sqcap m_3)$. Consequently, $(m_1 \sqcap m_3) \sqcup (m_2 \sqcap m_3)$ always exists. So, if $(m_1 \sqcup m_2) \sqcap m_3$ exists, then $(m_1 \sqcup m_2) \sqcap m_3 = (m_1 \sqcap m_3) \sqcup (m_2 \sqcap m_3)$, since a Brouwerian algebra is distributive.

($d$) This follows directly from the corresponding property of Brouwerian algebra.

($e$) This follows from the definition of Brouwerian algebra which requires that $(a \sqcup b) \sqsupseteq c$ iff $b \sqsupseteq (c \dot{-} a)$. □

### 4.2. Relating the Partial Ordering to Function Composition

We chose to define ≤ as in Definition 4.1 because it reveals the structure of the partial order immediately. Our interest in this partial order, however, stems from its relation to function composition, which will be presented soon as Proposition 4.6. We begin with one of the important properties satisfied by the monoid of modification functions $(M, \circ, I)$.

**Proposition 4.4.** (Extended idempotence). $m_1 \circ m_2 \circ m_1 = m_1 \circ m_2$, for all $m_1, m_2 \in M$.

**Proof.** Let $m_1$ be $<x_1, y_1>$ and $m_2$ be $<x_2, y_2>$. Then,

$$\begin{aligned}
<x_1,y_1> \circ <x_2,y_2> \circ <x_1,y_1> &= <x_1 \sqcup (x_2 \sqcap y_1),\ y_1 \sqcap (y_2 \sqcup x_1)> \circ <x_1,y_1> \\
&= <x_1 \sqcup (x_2 \sqcap y_1) \sqcup (x_1 \sqcap (\cdots)),\ y_1 \sqcap (y_2 \sqcup x_1) \sqcap (y_1 \sqcup (\cdots))> \\
&= <x_1 \sqcup (x_2 \sqcap y_1),\ y_1 \sqcap (y_2 \sqcup x_1)> \\
&= <x_1,y_1> \circ <x_2,y_2> \ \square
\end{aligned}$$

**Corollary 4.5.** $\circ$ is idempotent.

**Proof.** Let $m_2$ be the identity function in Proposition 4.4. □

**Proposition 4.6.**

($a$) $m_1 \le m_2$ iff $m_1 \circ m_2 = m_2$

($b$) $m_1 \le m_2$ iff $m_1 \circ m_2 = m_2 = m_2 \circ m_1$

($c$) $m_1 \le m_2$ iff $\exists m.\ m_2 = m_1 \circ m$.

($d$) $m_1 \circ m_2 \ge m_1$.

**Proof.**

($a$) Let $m_1$ be $<x_1,y_1>$ and let $m_2$ be $<x_2,y_2>$. $<x_1,y_1> \circ <x_2,y_2> = <x_2,y_2>$ iff $<x_1 \sqcup (x_2 \sqcap y_1), y_1 \sqcap (y_2 \sqcup x_1)> = <x_2,y_2>$ iff $x_1 \sqcup (x_2 \sqcap y_1) = x_2$ and $y_1 \sqcap (y_2 \sqcup x_1) = y_2$ iff $x_1 \sqsubseteq x_2$ and $y_1 \sqsupseteq y_2$ (recall that $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$, since for any modification $<x,y>$, $x \sqsubseteq y$) iff $<x_1,y_1> \leq <x_2,y_2>$.

($b$) If $m_1 \circ m_2 = m_2 = m_2 \circ m_1$ then $m_1 \leq m_2$ trivially (using ($a$)). Conversely, assume that $m_1 \leq m_2$, i.e., $m_1 \circ m_2 = m_2$. Then, $m_2 \circ m_1 = (m_1 \circ m_2) \circ m_1 = m_1 \circ m_2$ (using Proposition 4.4) $= m_2$.

($c$) If $m_1 \leq m_2$ then $m_2 = m_1 \circ m_2$ (using ($a$)). Hence, $\exists m.\ m_2 = m_1 \circ m$ is true. Conversely, assume $m_2 = m_1 \circ m$ for some $m$. Then, $m_1 \circ m_2 = m_1 \circ m_1 \circ m = m_1 \circ m = m_2$. Hence $m_1 \leq m_2$ (using ($a$)).

($d$) Follows from ($c$).

□

The above proposition justifies interpreting $\leq$ as capturing the notion of subsumption among modifications. Thus, modification $m_1$ is subsumed by (or "is contained in", "is smaller than") modification $m_2$ if performing $m_1$ and $m_2$ in either order does nothing more than performing $m_2$.

### 4.3. Compatibility Among Modifications

Our next step is to introduce a notion of *conflict* among modifications and relate it to the existence of upper bounds. Figure 4.3 provides some motivation for the following definition. Programs *a* and *b* are obtained by performing modifications $m_1$ and $m_2$, respectively, to program *base*. If $m_1$ and $m_2$ commute (with respect to function composition), it seems reasonable that the result of integrating *a* and *b* with respect to *base* should be the program $(m_1 \circ m_2)(base)$. We will subsequently see in Proposition 4.16 that if $m_1 \circ m_2 \neq m_2 \circ m_1$, then $(m_1 \circ m_2)(base) \neq (m_2 \circ m_1)(base)$, for any *base*. (Sections 5.3 and 5.4 will address the question of what integration should yield if $(m_1 \circ m_2)(base) \neq (m_2 \circ m_1)(base)$.)

**Definition 4.7.** Modifications $m_1$ and $m_2$ are said to *conflict* or *be incompatible* if $m_1 \circ m_2 \neq m_2 \circ m_1$. Otherwise, they are said to *commute* or *be compatible* with each other.

Thus, we use the phrase "are compatible" as a synonym for the phrase "commute with respect to function composition," essentially because it clarifies the intuition behind various

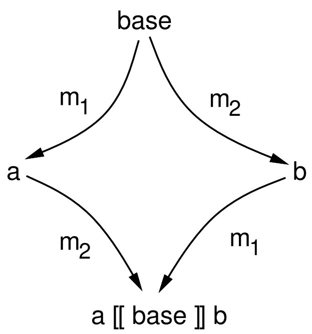


**Figure 4.3.** Integration by composing modifications.

of the following propositions.

**Proposition 4.8.**

($a$) $m_1$ and $m_2$ are compatible iff $m_1$ and $m_2$ have a least upper bound in which case $m_1 \sqcup m_2 = m_1 \circ m_2$.

($b$) If $m_3 \le m_1$ and $m_4 \le m_2$ and if $m_1$ and $m_2$ are compatible, then $m_3$ and $m_4$ are compatible. (Thus, compatibility is preserved downwards: parts of compatible modifications are themselves compatible.)

**Proof.**

($a$) We first show that if $m_1$ and $m_2$ are compatible, then $m_1$ and $m_2$ have an upper bound. Since $m_1$ and $m_2$ are compatible, we have, by definition, $m_1 \circ m_2 = m_2 \circ m_1$. We know from Proposition 4.6($d$) that $m_1 \circ m_2 \ge m_1$. Similarly, $m_1 \circ m_2 = m_2 \circ m_1 \ge m_2$. Thus, $m_1 \circ m_2$ is an upper bound of $m_1$ and $m_2$. Hence, from Proposition 4.3($a$), $m_1$ and $m_2$ have a least upper bound.

We now show that if $m_1$ and $m_2$ have a least upper bound, they commute. Let $m_1$ be $<x_1, y_1>$ and let $m_2$ be $<x_2, y_2>$. If $m_1$ and $m_2$ have a least upper bound then $x_1 \sqsubseteq y_2$ and $x_2 \sqsubseteq y_1$ (See Proposition 4.2($b$).) Hence, $m_1 \circ m_2 = <x_1 \sqcup (x_2 \sqcap y_1), y_1 \sqcap (y_2 \sqcup x_1)> = <x_1 \sqcup x_2, y_1 \sqcap y_2> = m_1 \sqcup m_2$. Similarly, $m_2 \circ m_1 = <x_2 \sqcup x_1, y_2 \sqcap y_1> = m_2 \sqcup m_1$. Therefore, $m_1$ and $m_2$ commute. The result follows.

($b$) If $m_1$ and $m_2$ are compatible, then they have an upper bound, which is also an upper bound for $m_3$ and $m_4$. Hence, from ($a$) and 4.3($a$), $m_3$ and $m_4$ are compatible. □

*Remark*. Let us now consider what it means for two modifications $<x_1, y_1>$ and $<x_2, y_2>$ to conflict. We have seen that the two modifications have a least upper bound iff $x_1 \sqsubseteq y_2$ and $x_2 \sqsubseteq y_1$ (Proposition 4.2(*b*)). Thus, two modifications are compatible iff whatever program slices one modification adds are preserved (*i.e.*, not deleted) by the other modification. For example, consider a program *base*, which contains a slice *s*. Let $m_1$ be a modification that adds a new slice $s'$ to *base*, where $s'$ “makes use” of existing slice *s* (that is, *s* is a subslice of $s'$), and let $m_2$ be a modification that deletes slice *s*. Then, $m_1$ and $m_2$ will be incompatible. □

We now establish some results relating to the monotonicity of $\circ$.

**Proposition 4.9.**

(*a*) If $m_1 \geq m_2$ then $m \circ m_1 \geq m \circ m_2$.

(*b*) If $m_1 \geq m_2$ and $m_1$ and $m$ commute, then $m_1 \circ m \geq m_2 \circ m$.

(*c*) If $m_1 \leq m_2 \leq m_3$ and $m_1 \circ m \leq m_3 \circ m$, then $m_1 \circ m \leq m_2 \circ m \leq m_3 \circ m$.

**Proof.**

(*a*) Assume that $m_1 \geq m_2$. Then, $(m \circ m_2) \circ (m \circ m_1) = m \circ m_2 \circ m_1 = m \circ m_1$. Hence, $m \circ m_1 \geq m \circ m_2$ by Proposition 4.6(*a*).

(*b*) Assume that $m_1 \geq m_2$ and that $m_1$ and $m$ commute. Then, it follows that $m_2$ and $m$ commute. Hence, from (*a*), $m_1 \circ m = m \circ m_1 \geq m \circ m_2 = m_2 \circ m$.

(*c*) We have $m_1 \circ m \leq m_3 \circ m$ and $m_2 \leq m_3 \leq m_3 \circ m$. Since $m_1 \circ m$ and $m_3$ have an upper bound, they commute. (Propositions 4.8(*a*) and 4.3(*b*).) Similarly, $m_1 \circ m$ and $m_2$ commute. Hence, $m_2 \circ m = (m_2 \circ m_1) \circ m = m_2 \circ (m_1 \circ m) = (m_1 \circ m) \circ m_2 \geq m_1 \circ m$. Thus we have $m_2 \circ m \geq m_1 \circ m$.

Since $m_3$ commutes with both $m_1 \circ m$ and $m_2$, it commutes with $m_2 \circ (m_1 \circ m)$, *i.e.* $m_2 \circ m$. Hence, $m_3 \circ m = (m_3 \circ m_2) \circ m = m_3 \circ (m_2 \circ m) = (m_2 \circ m) \circ m_3 \geq m_2 \circ m$. Thus we get, $m_1 \circ m \leq m_2 \circ m \leq m_3 \circ m$. □

If two modifications $m_1$ and $m_2$ are incompatible, it is natural to ask if we can decompose $m_1$ into two parts, a part that is compatible with $m_2$, and the remaining part. We now define a binary operation $|_c$ such that $m_1 |_c m_2$ may be interpreted as that part of modification $m_1$ that is *compatible* with modification $m_2$.

**Definition 4.10.** Define the binary operation $|_c$ on M as follows:

$$m_1 |_c m_2 =_{def} (m_2 \circ m_1) \sqcap m_1.$$

*Remark*. In terms of Brouwerian algebraic operators, the above definition yields $<x_1, y_1> |_c <x_2, y_2> = <x_1 \sqcap y_2, y_1 \sqcup x_2>$. It may be observed that $m_1 |_c m_2$ adds those program components that $m_1$ adds and $m_2$ does not delete, and that $m_1 |_c m_2$ deletes those program components that $m_1$ deletes and $m_2$ does not add. Thus, $m_1 |_c m_2$ does represent, in an intuitive sense, the part of $m_1$ that is compatible with $m_2$. □

We can show that $m_1 |_c m_2$ represents the part of $m_1$ that is compatible with $m_2$ in a more formal sense too. More generally, assume we want to identify that part of modification $m_1$ that satisfies a property (predicate) $P$. The set $\{ m \le m_1 \mid P(m) \}$ is the set of all modifications that are part of $m_1$ and satisfy property $P$. Thus, either $\max \{ m \le m_1 \mid P(m) \}$ or $\min \{ m \le m_1 \mid P(m) \}$, depending on the property $P$, is the natural choice as *the* part of $m_1$ that satisfies property $P$ (provided it exists). For instance, while it is reasonable to think of $\max \{ m \le m_1 \mid m$ is compatible with $m_2 \}$ as the part of $m_1$ that is compatible with $m_2$, it is $\min \{ m \le m_1 \mid m$ is not compatible with $m_2 \}$ that is appropriate as the part of $m_1$ that is not compatible with $m_2$. This is because the property of being compatible with modification $m_2$ is preserved downwards—that is, if $m$ is compatible with $m_2$ and $m' \le m$, then $m'$ is compatible with $m_2$—while the property of being incompatible with $m_2$ is preserved upwards.

**Proposition 4.11.**

($a$) $m_1 |_c m_2$ is compatible with $m_2$.

($b$) $m_1 |_c m_2 \le m_1$.

($c$) $m_1 |_c m_2 = \max \{ m \le m_1 \mid m$ is compatible with $m_2 \}$.

($d$) $m_1 |_c m_2$ and $m_2 |_c m_1$ are compatible with each other.

**Proof.**

($a$) $m_1 |_c m_2 \le m_2 \circ m_1$, from the definition of $|_c$. $m_2 \le m_2 \circ m_1$, from Proposition 4.6($d$). Since $m_1 |_c m_2$ and $m_2$ have an upper bound (namely, $m_2 \circ m_1$) it follows from Propositions 4.8($a$) and 4.3($a$) that they are compatible.

($b$) Follows from the definition of $|_c$.

($c$) Let $A$ denote the set $\{ m \le m_1 \mid m$ is compatible with $m_2 \}$. From ($a$) and ($b$), it follows that $m_1 |_c m_2 \in A$. Consider any $m \in A$. Then, $m \le m_1$. Hence, $m_2 \circ m \le m_2 \circ m_1$, from Proposition 4.9($a$). Hence, $(m_2 \circ m) \sqcap m \le (m_2 \circ m_1) \sqcap m \le (m_2 \circ m_1) \sqcap m_1 = m_1 |_c m_2$.

But $(m_2 \circ m) \sqcap m$ is $m$, since $m_2 \circ m$ is $m_2 \sqcup m$ whenever $m_2$ and $m$ are compatible. Hence, $m \le m_1 |_c m_2$.

($d$) $m_1 |_c m_2$ is compatible with $m_2$, from ($a$). $m_2 |_c m_1 \le m_2$, from ($b$). Hence, $m_1 |_c m_2$ is compatible with $m_2 |_c m_1$, from Proposition 4.8($b$). □

The following equalities are useful in manipulating and rewriting terms. Their intuitive meaning is discussed following the proof of the equations.

**Proposition 4.12.**

($a$) $m_1 \circ m_2 = (m_1 \dot{-} m_2) \circ m_2$.

($b$) $(m_1 \circ m_2) \dot{-} m_2 = m_1 \dot{-} m_2$.

($c$) $m_1 \circ m_2 = (m_2 |_c m_1) \circ m_1 = (m_2 |_c m_1) \sqcup m_1$.

($d$) If $m_1 \circ m \ge m_2$, then $m \ge m_2 \dot{-} m_1$.

($e$) If $m_1 \circ m \ge m_2$, then $m_1 \ge m_2 \dot{-} m$.

**Proof.**

($a$) $m_1 = (m_1 \dot{-} m_2) \sqcup (m_1 \sqcap m_2)$ (see Proposition 4.3($b$)) $= (m_1 \dot{-} m_2) \circ (m_1 \sqcap m_2)$. Therefore, $m_1 \circ m_2 = (m_1 \dot{-} m_2) \circ (m_1 \sqcap m_2) \circ m_2 = (m_1 \dot{-} m_2) \circ m_2$, since $m_1 \sqcap m_2 \le m_2$.

($b$) Let $m_1$ be $<x_1, y_1>$ and let $m_2$ be $<x_2, y_2>$. Then, $(<x_1, y_1> \circ <x_2, y_2>) \dot{-} <x_2, y_2> = <x_1 \sqcup (x_2 \sqcap y_1), y_1 \sqcap (y_2 \sqcup x_1)> \dot{-} <x_2, y_2> = <(x_1 \sqcup (x_2 \sqcap y_1)) \dot{-} x_2, (y_1 \sqcap (y_2 \sqcup x_1)) \div y_2> = <x_1 \dot{-} x_2, y_1 \div y_2> = <x_1, y_1> \dot{-} <x_2, y_2>$.

($c$) $m_1 \circ m_2 = (m_1 \circ m_2) \sqcup m_1 = ((m_1 \circ m_2) \sqcap m_2) \sqcup ((m_1 \circ m_2) \dot{-} m_2) \sqcup m_1 = (m_2 |_c m_1) \sqcup m_1$. The last step uses the fact that $(m_1 \circ m_2) \dot{-} m_2 \le m_1$, which follows from ($b$).

($d$) From ($c$) $m_1 \circ m = (m |_c m_1) \sqcup m_1$. It follows from the hypothesis $m_1 \circ m \ge m_2$ that $(m |_c m_1) \sqcup m_1 \ge m_2$. Hence, from Proposition 4.3($e$), $m |_c m_1 \ge m_2 \dot{-} m_1$. From Proposition 4.11($b$) $m \ge m |_c m_1$, and the result follows.

($e$) From ($c$) $m_1 \circ m = (m |_c m_1) \sqcup m_1$. It follows from the hypothesis $m_1 \circ m \ge m_2$ that $(m |_c m_1) \sqcup m_1 \ge m_2$. Hence, from Proposition 4.3($e$), $m_1 \ge m_2 \dot{-} (m |_c m_1)$. From Proposition 4.11($b$) $m \ge m |_c m_1$, and hence, $m_2 \dot{-} (m |_c m_1) \ge m_2 \dot{-} m$. The result follows. □

The first equation essentially follows from the idempotence of modifications. If modification $m_1$ is applied after modification $m_2$, the effect is the same as if $m_1 \dot{-} m_2$ were applied (after $m_2$), since the part of $m_1$ that is common to $m_2$ (namely, $m_1 \sqcap m_2$) achieves

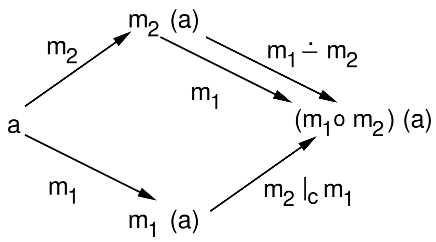


**Figure 4.4.** Relation between $\circ$, $\dot{-}$ and $|_c$.

nothing (that $m_2$ has not already). The third equation, on the other hand, says that when $m_1$ is performed after $m_2$, the "only effects of $m_2$ that survive" are those that are compatible with $m_1$. These two properties are summarized by the commutative diagram of Figure 4.4. The fourth and fifth inequalities are generalizations of the following Brouwerian law: if $m_1 \sqcup m \geq m_2$, then $m \geq m_2 \dot{-} m_1$.

We have so far looked at the set $M$ of modifications. We now take a look at certain special subsets of $M$. In particular, we will consider the set of all modifications that yield $a$ when applied to $b$, for specific programs $a$ and $b$.

**Definition 4.13.** For any $a$ and $b$ in P, let $M_{a,b}$ denote the set of modifications that map $b$ to $a$. Thus,

$$M_{a,b} =_{def} \{ m \in M \mid m(b) = a \}.$$

Observe that $M_{a,b}$ may be thought of as the collection of morphisms from $b$ to $a$ in the category $\mathcal{C}$. More precisely, it is the set of modification functions associated with this set of morphisms.

**Proposition 4.14.** $M_{a,b} = \{ m \mid \langle a \dot{-} b, a \div b \rangle \leq m \leq \langle a,a \rangle \}$. In particular, $M_{a,b}$ has a least element, namely $\langle a \dot{-} b, a \div b \rangle$, and a greatest element, namely $\langle a,a \rangle$.

**Proof.** Note that $\langle x,y \rangle(b) = (b \sqcap y) \sqcup x = (b \sqcup x) \sqcap y$ (since $x \sqsubseteq y$).

$\langle x,y \rangle \in M_{a,b} \Leftrightarrow (b \sqcap y) \sqcup x = a$ (and $(b \sqcup x) \sqcap y = a$)

$\Leftrightarrow b \sqcap y \sqsubseteq a$ and $x \sqsubseteq a$ and $b \sqcup x \sqsupseteq a$ and $y \sqsupseteq a$

$\Leftrightarrow y \sqsubseteq a \div b$ and $x \sqsubseteq a$ and $x \sqsupseteq a \dot{-} b$ and $y \sqsupseteq a$

$\Leftrightarrow \langle x,y \rangle \geq \langle a \dot{-} b, a \div b \rangle$ and $\langle x,y \rangle \leq \langle a,a \rangle$ □

**Corollary 4.15.** $M_{a,a} = \{ m \mid m \sqsubseteq \langle a,a \rangle \}$.

**Proof.**

This follows immediately, since $\langle a \dot{-} a, a \div a \rangle = I$, the least element of $M$. □

**Proposition 4.16.** Let $a$ be some program, and $m_1$ and $m_2$ be two modifications. Then, $(m_1 \circ m_2)(a) = (m_2 \circ m_1)(a)$ iff $m_1 \circ m_2 = m_2 \circ m_1$.

**Proof.** We need to show that if $(m_1 \circ m_2)(a) = (m_2 \circ m_1)(a)$ then $m_1 \circ m_2 = m_2 \circ m_1$. Let $(m_1 \circ m_2)(a) = (m_2 \circ m_1)(a) = b$. Then, $m_1(m_2(a)) = b$. Hence, $m_1 \leq \langle b,b \rangle$ (from Proposition 4.14). Similarly, $m_2 \leq \langle b,b \rangle$, since $m_2(m_1(a)) = b$. Hence both $m_1$ and $m_2$ have an upper bound, namely $\langle b,b \rangle$. Hence, from Proposition 4.8($a$), $m_1$ and $m_2$ commute. □

The above proposition says that if $m_1$ and $m_2$ conflict, then $(m_1 \circ m_2)(a) \neq (m_2 \circ m_1)(a)$, for any program $a$. This provides further justification for the definition of incompatibility we use (Definition 4.7). Recall the situation considered earlier. We have a program *base*, and programs $a$ and $b$ obtained by making modifications $m_1$ and $m_2$ respectively to *base*, and we are interested in integrating $a$ and $b$ with respect to *base*. The question that arises is whether $(m_1 \circ m_2)(base) = (m_2 \circ m_1)(base)$. The above proposition says that this question is equivalent to the question of whether $m_1 \circ m_2 = m_2 \circ m_1$, that is whether $m_1$ and $m_2$ commute.

## 5. Operations On Program Modifications

We now turn our attention to certain operations on modifications. We define the *fm*-algebra operators $\Delta$ and $+$, and a couple of related operators, and study their properties.

### 5.1. Inferring modifications

The idea behind the operator $\Delta$, as explained earlier, is as follows. We have programs $b$ and $a$, where $a$ was obtained by making some modification $m$ to $b$. We would like to infer modification $m$ from the programs $a$ and $b$. The value of $\Delta(a,b)$ is supposed to be this inferred modification. As we have already seen, in general, there is no unique modification that yields $a$ when applied to $b$. There is a whole set of such modifications, namely $M_{a,b}$. Note that $\langle a,a \rangle$, the greatest element in $M_{a,b}$, is nothing but the constant-valued function $\lambda z.a$. We define $\Delta(a,b)$, conservatively, to be the least modification that changes $b$ to $a$, namely $\langle a \dot{-} b, a \div b \rangle$. We will subsequently see that our inability to determine uniquely the modification the user intended in developing $a$ from $b$ does not greatly affect the result of integration. (See the remark after Proposition 6.4.)

**Definition 5.1.** $\Delta(a,b) =_{def} \min_{\leq}(M_{a,b})$

**Proposition 5.2.**

(*a*) $\Delta(a,b)(b) = a$.

(*b*) $\Delta(m(b),b) \leq m$.

(*c*) $\Delta(a,a) = I$.

(*d*) $\Delta(a,b) = <a \dot{-} b, a \div b> = <a,a> \dot{-} <b,b>$.

(*e*) $\Delta(m(b),b) = m \dot{-} <b,b>$.

**Proof.** (*a*) and (*b*) follow directly from the definition of $\Delta$. (*c*) is a special case of (*b*). (*d*) follows from Propositions 4.14 and 4.2(*c*). As for (*e*), let $a$ denote $m(b)$. Then, from Proposition 4.14, $\Delta(a,b) \leq m \leq <a,a>$. Hence, $\Delta(a,b) \dot{-} <b,b> \leq m \dot{-} <b,b> \leq <a,a> \dot{-} <b,b>$. That is, $\Delta(a,b) \leq m \dot{-} <b,b> \leq \Delta(a,b)$. Hence, the result follows. □

A morphism $m$ from an object $b$ to an object $a$ is said to be a *principal morphism* (see [1], for instance) if for every morphism $m_1$ from $b$ to $a$ there exists a morphism $m_2$ from $b$ to itself such that $m_1 = m \circ m_2$. (See Figure 5.5(*a*).) We now show that in category $\mathcal{C}$ the morphism from program $b$ to $a$ labelled $\Delta(a,b)$ is a principal morphism.

**Proposition 5.3.** The morphism $[b, \Delta(a,b), a]$ is a principal morphism.

**Proof.** See Figure 5.5(*b*). Consider any modification $m_1$ such that $m_1(b) = a$. We know from Proposition 4.3(*b*) that $m_1 = (m_1 \dot{-} <b,b>) \sqcup (m_1 \sqcap <b,b>)$. It follows from Proposition 5.2(*e*) that $m_1 \dot{-} <b,b>$ is $\Delta(a,b)$, while it follows from Corollary 4.15 that $m_1 \sqcap <b,b>$ is in $M_{b,b}$. The result follows. □

### 5.2. Combining modifications

Now we consider the *fm*-algebra operator $+$ that "combines" modifications. Our interpretation of the partial order $\leq$ on modifications is that $m \leq m'$ represents the fact that modification $m$ is contained as a part of modification $m'$. Thus, it is natural to expect $m_1 + m_2$ to be the join $m_1 \sqcup m_2$ of $m_1$ and $m_2$ with respect to the partial order $\leq$. But, we know that $\sqcup$ is a partial operator: not every pair of modifications have a least upper bound. We observed ear-

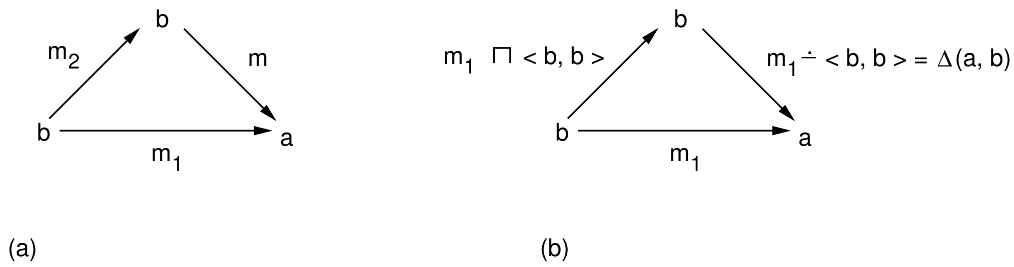


**Figure 5.5.** $(a)$ The definition of principal morphism. $(b)$ $\Delta(a,b)$ is the principal morphism from $b$ to $a$.

lier that two modifications do not have a least upper bound precisely when they do not commute: when one of the modifications “adds” some program component(s) that the other modification “deletes”. The HPR algorithm is based on the assumption that in such circumstances the addition of the program component should take precedence over the deletion. We may say that the HPR algorithm resolves the conflict in favor of the addition of program components. (See Example 2.5, for instance.) This reasoning leads to the following definition of $+$.

**Definition 5.4.** $<x_1,y_1> + <x_2,y_2> =_{def} <x_1 \sqcup x_2, (y_1 \sqcap y_2) \sqcup x_1 \sqcup x_2>$.

**Proposition 5.5.** $(M, +, I)$ is a join semi-lattice with $I$ as the least element. Thus,

$(a)$ $m + m = m$

$(b)$ $m_1 + m_2 = m_2 + m_1$

$(c)$ $(m_1 + m_2) + m_3 = m_1 + (m_2 + m_3)$

$(d)$ $m + I = m = I + m$

**Proof.** These follow from the corresponding properties of both $\sqcup$ and $\sqcap$, if we expand the above expressions into ordered pairs and use the definition of $+$. $\square$

However, note that $(M, +, I)$ is not a join semi-lattice with respect to the partial order $\leq$ we have used so far. The relationship between $+$ and $\leq$ will be clearer soon.

**Proposition 5.6.**

$(a)$ If $m_1$ and $m_2$ are compatible then $m_1 + m_2 = m_1 \circ m_2 = m_1 \sqcup m_2$.

$(b)$ $(m_1 |_c m_2) \sqcup (m_2 |_c m_1) \leq m_1 + m_2$.

**Proof.**

($a$) Let $m_1 = <x_1, y_1>$ and $m_2 = <x_2, y_2>$. If $m_1$ and $m_2$ are compatible then they have an upper bound, and hence $x_1 \sqcup x_2 \sqsubseteq y_1 \sqcap y_2$. Hence, $m_1 + m_2 = <x_1 \sqcup x_2, y_1 \sqcap y_2> = m_1 \circ m_2 = m_1 \sqcup m_2$.

($b$) Let $m_1 = <x_1, y_1>$ and $m_2 = <x_2, y_2>$. Note that $m_1 |_c m_2 = <x_1 \sqcap y_2, y_1 \sqcup x_2>$, and $(m_1 |_c m_2) \sqcup (m_2 |_c m_1) = <(x_1 \sqcup x_2) \sqcap (y_1 \sqcap y_2), (y_1 \sqcap y_2) \sqcup (x_1 \sqcup x_2)>$. Hence, the result follows. □

The above proposition summarizes the previous discussion about the $+$ operator. When $m_1 \sqcup m_2$ exists, it is equal to $m_1 + m_2$. Thus, $+$ may be regarded as a suitable totalization of the partial operation $\sqcup$. The operator $+$ itself satisfies the properties of a semilattice join operator. The corresponding partial order may be viewed as a refinement of the partial order $\leq$. The second proposition shows that $m_1 + m_2$ contains both $m_1 |_c m_2$ and $m_2 |_c m_1$, as expected intuitively. But, the inequality implies that $m_1 + m_2$ is not obtained by simply putting together the parts of $m_1$ and $m_2$ that are compatible with $m_2$ and $m_1$ respectively. Rather, as explained earlier, we may think of $m_1 + m_2$ as being obtained by putting together those parts of $m_1$ and $m_2$ that are not overruled by $m_2$ and $m_1$ respectively. This raises the question of what it means for a modification to *overrule* another. We utilize the $+$ operator to formalize this notion in the following section.

### 5.3. Resolving incompatibility among modifications

**Definition 5.7.** Modification $m_2$ is said to *overrule part of* modification $m_1$ if $m_2 + m_1 \not\geq m_1$.

Note that the condition $m_2 + m_1 \not\geq m_1$ says that when modifications $m_2$ and $m_1$ are combined, the resulting combined modification $m_2 + m_1$ does not completely contain modification $m_1$. We now turn our attention to identifying the part of a modification $m_1$ that is not overruled by another modification $m_2$, denoted by $m_1 |_o m_2$.

**Definition 5.8.** Define the two binary operators $|_o$ and $\dot{-}_o$ as follows:

$$m_1 |_o m_2 =_{def} (m_1 + m_2) \sqcap m_1$$

$$m_1 \dot{-}_o m_2 =_{def} (m_1 |_o m_2) \dot{-} m_2$$

**Proposition 5.9.**

($a$) $m_2$ overrules part of $m_1$ iff $m_1 |_o m_2 \neq m_1$.

($b$) $(m_1 |_o m_2) |_o m_2 = m_1 |_o m_2$

($c$) $m_1 |_o m_2$ and $m_1 \dot{-}_o m_2$ are monotonic in $m_1$ and anti-monotonic in $m_2$.

($d$) $m_1 |_o m_2 = \max_{\leq} \{ m \leq m_1 \mid m_2$ does not overrule part of $m \}$.

($e$) If $m_1 \leq m_2$ and $m$ overrules part of $m_1$ then $m$ overrules part of $m_2$.

**Proof.**

($a$) Since $\geq$ is a partial ordering, it follows that $m_2 + m_1 \geq m_1$ iff $(m_2 + m_1) \sqcap m_1 = m_1$, that is, iff $m_1 |_o m_2 = m_1$. In other words, $m_2 + m_1 \not\geq m_1$ iff $m_1 |_o m_2 \neq m_1$.

($b$) The definition of $|_o$ expands to $<x_1, y_1> |_o <x_2, y_2> = <x_1, y_1 \sqcup x_2>$. Consequently, $(m_1 |_o m_2) |_o m_2 = m_1 |_o m_2$.

($c$) It can be verified from the definition that $<x_1, y_1> |_o <x_2, y_2>$ simplifies to $<x_1, y_1 \sqcup x_2>$. The result follows.

($d$) Let $A$ denote the set $\{ m \leq m_1 \mid m_2$ does not overrule part of $m \}$. From ($a$) and ($b$) it follows that $m_1 |_o m_2 \in A$. Consider any $m \in A$. Then, $m \leq m_1$. Hence, from ($c$), $m |_o m_2 \leq m_1 |_o m_2$. But, using ($a$), $m = m |_o m_2$ since $m_2$ does not overrule part of $m$. Hence, $m \leq m_1 |_o m_2$. The result follows.

($e$) Note that $<x, y>$ does not overrule part of $<x_2, y_2>$ iff $x \sqsubseteq y_2$. Thus, if $<x, y>$ does not overrule $<x_2, y_2>$ and $<x_1, y_1> \leq <x_2, y_2>$, it follows that $<x, y>$ does not overrule $<x_1, y_1>$. □

The following proposition concerns different ways of looking at the expression $m_1 + m_2$. Some of these results are summarized by the commutative diagram shown in Figure 5.6.

**Proposition 5.10.**

($a$) $m_1 |_c m_2 \leq m_1 |_o m_2 \leq m_1$

($b$) $m_1 |_o m_2$ and $m_2 |_o m_1$ commute.

($c$) $m_1 + m_2 = (m_1 |_o m_2) \sqcup (m_2 |_o m_1)$

($d$) $m_1 |_o m_2 = (m_1 \dot{-}_o m_2) \sqcup (m_1 \sqcap m_2)$

($e$) $m_1 + m_2 = (m_1 \dot{-}_o m_2) \sqcup (m_1 \sqcap m_2) \sqcup (m_2 \dot{-}_o m_1)$

($f$) $m_1 + m_2 = (m_1 |_o m_2) \circ m_2 = (m_2 |_o m_1) \circ m_1$

($g$) $m_1 + m_2 = (m_1 \dot{-}_o m_2) \circ m_2 = (m_2 \dot{-}_o m_1) \circ m_1$

**Proof.**

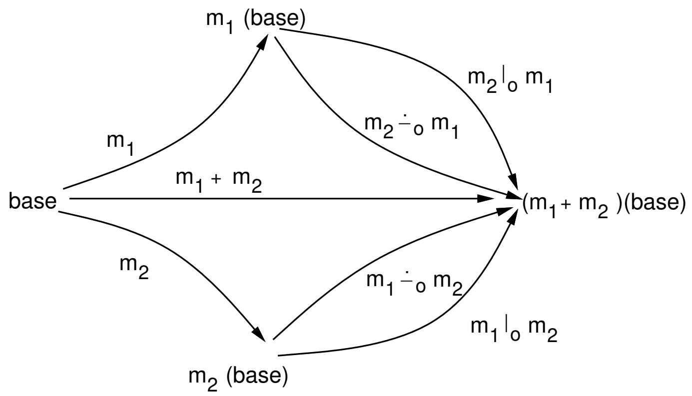


**Figure 5.6.** Relation between $|_o$, $\dot{-}_o$ and $+$.

($a$) $m_1 |_o m_2 \le m_1$ from the definition of $|_o$. It follows from Proposition 5.6($b$) that $m_1 |_c m_2 \le m_1 + m_2$. Since $m_1 |_c m_2 \le m_1$ also, it follows that $m_1 |_c m_2 \le (m_1 + m_2) \sqcap m_1 = m_1 |_o m_2$.

($b$) From the definition of $|_o$, it follows that $m_1 + m_2$ is an upper bound for both $m_1 |_o m_2$ and $m_2 |_o m_1$. Hence, it follows from Propositions 4.8($a$) and 4.3($a$) that $m_1 |_o m_2$ and $m_2 |_o m_1$ commute.

($c$) We verify this proposition by expanding the terms into ordered pairs. Let $m_1$ be $<x_1, y_1>$ and let $m_2$ be $<x_2, y_2>$. Then, $(m_1 |_o m_2) \sqcup (m_2 |_o m_1) = <x_1, y_1 \sqcup x_2> \sqcup <x_2, y_2 \sqcup x_1> = <x_1 \sqcup x_2,\ (y_1 \sqcup x_2) \sqcap (y_2 \sqcup x_1)> = <x_1 \sqcup x_2,\ (y_1 \sqcap y_2) \sqcup (x_2 \sqcap y_2) \sqcup (y_1 \sqcap x_1) \sqcup (x_2 \sqcap x_1)> = <x_1 \sqcup x_2,\ (y_1 \sqcap y_2) \sqcup (x_1 \sqcup x_2)>$ (since $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$) $= m_1 + m_2$ by definition.

($d$) $m_1 |_o m_2 = (m_1 |_o m_2) \dot{-} m_2 \sqcup (m_1 |_o m_2) \sqcap m_2$ from 4.3($b$). The result follows since $(m_1 |_o m_2) \sqcap m_2 = m_1 \sqcap m_2$.

($e$) This follows from ($c$) and ($d$) above.

($f$) Let $m_1$ be $<x_1, y_1>$ and let $m_2$ be $<x_2, y_2>$. Then, $(m_1 |_o m_2) \circ m_2 = <x_1, y_1 \sqcup x_2> \circ <x_2, y_2> = <x_1 \sqcup x_2, (y_1 \sqcup x_2) \sqcap (y_2 \sqcup x_1)> = m_1 + m_2$ (since $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$).

($g$) This follows from ($f$) and Proposition 4.12($a$). $\square$

## 6. Program Integration

We now look at some properties of the integration operator $\_[\![\_]\!]\_$ of the *fm*-algebra $(P, M, \Delta, +)$. Recall that the integration operator of an *fm*-algebra is defined as follows (see Definitions 3.2 and 3.3):

$$a[\![base]\!]b =_{def} (\Delta(a, base) + \Delta(b, base))(base).$$

Since $P$ is a Brouwerian algebra there is also the integration operator $\_[\_]\_$ defined on $P$ (see Definition 2.3):

$$a[base]b =_{def} (a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base).$$

These two definitions yield the same ternary operator.

**Theorem 6.1.** [9] $a[\![base]\!]b = a[base]b$.

**Proof.**

$$
\begin{aligned}
a[\![base]\!]b &= (\Delta(a, base) + \Delta(b, base))(base) \\
&= (\langle a \dot{-} base, a \div base \rangle + \langle b \dot{-} base, b \div base \rangle)(base) \\
&= (a \dot{-} base) \sqcup ((a \div base) \sqcap base \sqcap (b \div base)) \sqcup (b \dot{-} base) \\
&\qquad \text{(using the definition of } +\text{, and some simplification)} \\
&= (a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base) \\
&\qquad \text{(using the identity } (x \div y) \sqcap y = x \sqcap y\text{)} \\
&= a[base]b
\end{aligned}
$$

□

The above theorem, in conjunction with the results in [14], which relate the Brouwerian integration operator $\_[\_]\_$ to a variant of the HPR integration algorithm, establishes that the *fm*-algebra $(P, M, \Delta, +)$ models the HPR integration algorithm. Figure 3.1 illustrates various values in this *fm*-algebra.

**Proposition 6.2.**

(*a*) $a[\![base]\!]b = ((\Delta(a, base) \mid_o \Delta(b, base)) \sqcup (\Delta(b, base) \mid_o \Delta(a, base)))(base)$.

(*b*) $a[\![base]\!]b = ((\Delta(a, base) \dot{-}_o \Delta(b, base)) \sqcup (\Delta(a, base) \sqcap \Delta(b, base)) \sqcup (\Delta(b, base) \dot{-}_o \Delta(a, base)))(base)$.

(*c*) $a[\![base]\!]b = (\Delta(a, base) \mid_o \Delta(b, base))(b) = (\Delta(b, base) \mid_o \Delta(a, base))(a)$

(*d*) $a[\![base]\!]b = (\Delta(a, base) \dot{-}_o \Delta(b, base))(b) = (\Delta(b, base) \dot{-}_o \Delta(a, base))(a)$

(*e*) $\Delta(a[\![base]\!]b, base) \le \Delta(a, base) + \Delta(b, base)$.

($f$) $\Delta(a[\![base]\!]b, a) \leq \Delta(b, base) \dot{-}_o \Delta(a, base)$.

($g$) $\Delta(a[\![base]\!]b, a) = (\ \Delta(b, base)\,|_o\,\Delta(a, base)\ ) \dot{-} <a, a>$.

**Proof.**

($a$)-($d$) These follow immediately by rewriting $\Delta(a, base) + \Delta(b, base)$ in the definition of _[[_]]_ using Proposition 5.10. Thus, ($a$) follows from Proposition 5.10($c$). ($b$) follows from Proposition 5.10($e$). ($c$) follows from Proposition 5.10($f$). ($d$) follows from Proposition 5.10($g$).

($e$) This follows from the definition of $a[\![base]\!]b$ and Proposition 5.2($b$).

($f$) This follows from ($d$) and Proposition 5.2($b$).

($g$) This follows from ($c$) and Proposition 5.2($e$). □

As seen from the proof, most of the above properties are a consequence of Proposition 5.10. Some of these results are summarized by the commutative diagram in Figure 6.7, which is a special case of Figure 5.6.

**Proposition 6.3.** If $\Delta(a, base)$ and $\Delta(b, base)$ are compatible then

($a$) $a[\![base]\!]b = \Delta(a, base)(b) = \Delta(b, base)(a)$.

($b$) $\Delta(a[\![base]\!]b, base) = \Delta(a, base) + \Delta(b, base)$.

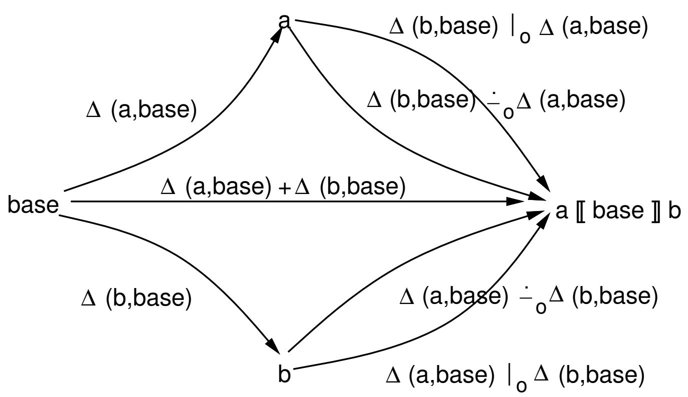


**Figure 6.7.** Relation between $|_o$, $\dot{-}_o$, and _[[_]]_.

**Proof.**

($a$) This follows as a special case of Proposition 6.2($c$) since if $\Delta(a, base)$ and $\Delta(b, base)$ are compatible then $\Delta(a, base) \ |_o \ \Delta(b, base) = \Delta(a, base)$ from Proposition 5.10($a$).

($b$) It follows from Proposition 5.2($e$) that $\Delta(a[[base]]b, base) = (\Delta(a, base) + \Delta(b, base)) \dot{-} <base, base>$. If $\Delta(a, base)$ and $\Delta(b, base)$ are compatible, then $\Delta(a, base) + \Delta(b, base) = \Delta(a, base) \sqcup \Delta(b, base) = <a \dot{-} base, a \div base> \sqcup <b \dot{-} base, b \div base> = <(a \dot{-} base) \sqcup (b \dot{-} base), (a \div base) \sqcap (b \div base)> = <(a \sqcup b) \dot{-} base, (a \sqcap b) \div base>$. Hence, $(\Delta(a, base) + \Delta(b, base)) \dot{-} <base, base>$ simplifies to $\Delta(a, base) + \Delta(b, base)$. The result follows. □

**Proposition 6.4.** Let $m_1$ and $m_2$ be compatible modifications. Then, $(m_1 \sqcup m_2)(base) = m_1(base)[[base]]m_2(base)$.

**Proof.** Let $a = m_1(base)$ and $b = m_2(base)$. Now, $\Delta(a, base) \le m_1$ and $\Delta(b, base) \le m_2$. Thus, $\Delta(a, base)$ and $\Delta(b, base)$ are themselves compatible with each other, and with $m_1$ and $m_2$ too. Then, $(m_1 \sqcup m_2)(base) = (m_1 \circ m_2)(base) = m_1(m_2(base)) = m_1(b) = m_1(\Delta(b, base)(base)) = (m_1 \circ \Delta(b, base))(base) = (\Delta(b, base) \circ m_1)(base) = \Delta(b, base)(m_1(base)) = \Delta(b, base)(a) = a[[base]]b$. The last step follows from Proposition 6.3($a$). □

*Remark*: The above proposition implies the following. Consider the integration of variants $a$ and $b$ of program *base*, where programs $a$ and $b$ are obtained by performing modifications $m_1$ and $m_2$, respectively, to program *base*, and $m_1$ and $m_2$ are compatible. Then the result of the integration is $(m_1 \circ m_2)$ $(base) = (\Delta(a, base) \circ \Delta(b, base))(base)$. This holds true irrespective of whether $\Delta(a, base)$ equals $m_1$ or $\Delta(b, base)$ equals $m_2$. Thus, we see that our inability to determine uniquely the modification the user intended in developing $a$ from *base* does not affect the result of integration in such cases. However, if the modifications $m_1$ and $m_2$ are incompatible, then the above result does not hold in general. The integration operation uses an optimistic approach: by choosing $\Delta(a, base)$ and $\Delta(b, base)$ to be the least elements of $M_{a,base}$ and $M_{b,base}$, respectively, it minimizes the chances of incompatibility among $\Delta(a, base)$ and $\Delta(b, base)$.

**7. Separating Consecutive Edits**

The problem of separating consecutive edits on some program into individual edits on the same program arises in the following context. Assume that an user starts with some program *base*, makes some changes to *base* to obtain program $a$, and then makes further changes to $a$ to obtain program $c$. The user then decides that the first set of changes made was a mistake, and that only the second set of changes was necessary and must be retained. Thus, the user desires a program that incorporates the second set of changes, but not the first, in the original program *base*.

Let us try to formalize the above problem in terms of the operators of *fm*-agebra. If the modification the user made to $a$ to obtain $c$ was $m$, then the program the user desires may be expressed as $m(base)$. But in separating consecutive edits, we have only the programs *base*, $a$, and $c$ available, and not $m$. Modification $m$ has to be inferred from the programs $a$ and $c$. This leads us to propose $\Delta(c,a)(base)$ as the solution (to this problem of separating consecutive edits).

However, the above approach is not the only one to the problem of separating consecutive edits. Other solutions have been previously proposed for this problem. The very first solution proposed was the "re-rooting" solution [6]: here, $c[\![a]\!]base$ is taken to be the desired program. A second solution previously proposed [13] was that the program we seek is a solution $x$ (called a "compatible integrand" with respect to *base*, $a$ and $c$) to the equation $a[\![base]\!]x = c$.

The solution we have proposed above is to "redo" the second modification on the original program. Instead, if we view the problem as that of "undoing" the first modification, we end up with yet another solution: $\Delta(base,a)(c)$. The assumption behind this solution is that we can use $\Delta(base,a)$ to undo the effect of $\Delta(a,base)$, which represents the first modification the user made. However, unless $a = base$, $\Delta(base,a)$ is not the functional inverse of $\Delta(a,base)$, that is, $\Delta(base,a) \circ \Delta(a,base) \neq I$, and hence we can, at best, view this solution as an approximate solution.

These various solutions to the problem of separating consecutive edits are all depicted in Figure 7.8. Our goal in this section is to show how the newly proposed solution, $\Delta(c,a)(base)$, relates to the previously proposed solutions and how the new solution is better.

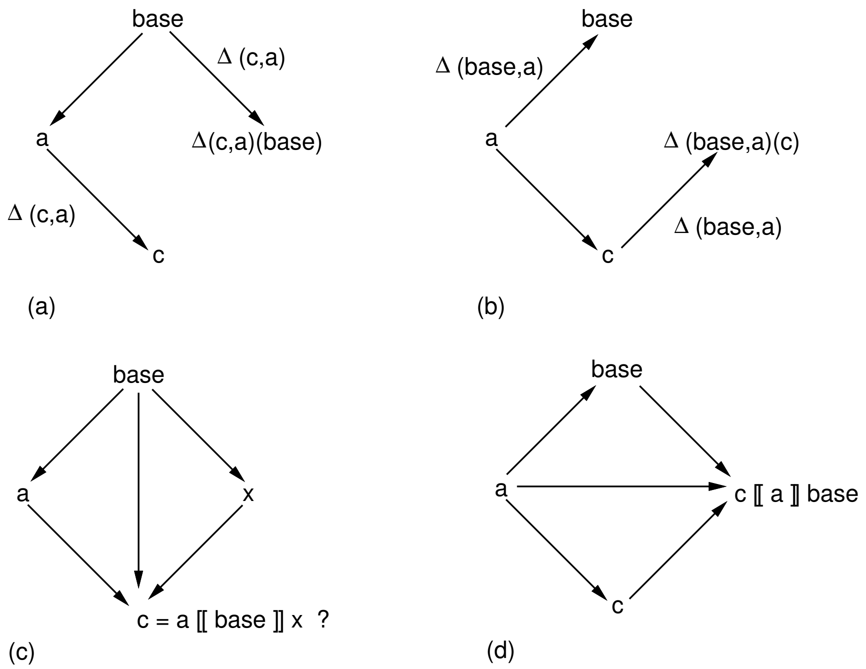


**Figure 7.8.** Different solutions to the problem of separating consecutive edits

### 7.1. $\Delta\,(c,a)(base)$ versus the compatible integrand approach

**Theorem 7.1.** If $\Delta\,(c,a)$ and $\Delta\,(a,base)$ do not conflict then $\Delta\,(c,a)(base)$ is a solution of the equation $a[\![base]\!]x = c$.

**Proof.** We show that $b = \Delta\,(c,a)(base)$ satisfies the given equation. Now, $\Delta\,(b,base) \le \Delta\,(c,a)$ (Proposition 5.2($b$)). Since $\Delta\,(c,a)$ and $\Delta\,(a,base)$ are compatible, it follows that $\Delta\,(b,base)$ and $\Delta\,(a,base)$ are also compatible (Proposition 4.8($b$)). Hence, from Proposition 6.3($a$), we get $a[\![base]\!]b = \Delta\,(a,base)(b) = \Delta\,(a,base)\,(\Delta\,(c,a)(base)) = \Delta\,(c,a)\,(\Delta\,(a,base)(base)) = c$. $\square$.

We now show that if the equation $a[\![base]\!]x = c$ has a solution for $x$ then $b = \Delta\,(c,a)(base)$ is a solution of that equation. Further, we show that $\Delta\,(c,a)$ is the *least* modification of *base* necessary to produce a solution of that equation (and thus $b$ is the "least-modified" solution of that equation) in the following sense: if $m\,(base)$ is any solution to the equation, then $m \ge \Delta\,(c,a) = \Delta\,(b,base)$.

**Theorem 7.2.** If the equation $a[\![base]\!]x = c$ has a solution for $x$ then $b = \Delta(c,a)(base)$ is a solution of that equation. Further, for any solution $x$ of the equation, $\Delta(x,base) \geq \Delta(b,base) = \Delta(c,a)$.

**Proof.** Let $x$ be such that $a[\![base]\!]x = c$. Let $b$ denote $\Delta(c,a)(base)$. We first show that $\Delta(b,base) = \Delta(c,a)$. $\Delta(b,base) = \Delta(c,a) \dot{-} <base,base>$, from Proposition 5.2($e$), and $\Delta(c,a) = (\Delta(x,base)\,|_o\,\Delta(a,base)) \dot{-} <a,a>$, from Proposition 6.2($g$). Thus, we get:

$$\begin{aligned}
\Delta(c,a) &= <x \dot{-} base, (x \div base) \sqcup (a \dot{-} base)> \;\dot{-}\; <a,a> \\
&= <(x \dot{-} base) \dot{-} a, [(x \div base) \sqcup (a \dot{-} base)] \div a> \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div a>
\end{aligned}$$

$$\begin{aligned}
\Delta(b,base) &= \Delta(c,a) \dot{-} <base,base> \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div a> \;\dot{-}\; <base,base> \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div a \div base> \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div (a \sqcap base)> \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div [(a \sqcap base) \sqcup (a \dot{-} base)]>
\end{aligned}$$

(making use of the Brouwerian identity $(y_1 \sqcup y_2) \div z = (y_1 \sqcup y_2) \div (z \sqcup y_2)$)

$$\begin{aligned}
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div a> \\
&= \Delta(c,a)
\end{aligned}$$

Now, $\Delta(c,a) \leq \Delta(x,base)\,|_o\,\Delta(a,base)$ (Proposition 6.2($f$)). Furthermore, $\Delta(x,base)\,|_o\,\Delta(a,base)$ is not overruled by $\Delta(a,base)$ (see Proposition 5.9$b$). Hence, $\Delta(c,a)$ is not overruled by $\Delta(a,base)$ (see Proposition 5.9$e$). Thus, $\Delta(c,a)\,|_o\,\Delta(a,base) = \Delta(c,a)$. Then, from Proposition 6.2($c$), we get $a[\![base]\!]b = (\,\Delta(b,base)\,|_o\,\Delta(a,base)\,)(a) = (\,\Delta(c,a)\,|_o\,\Delta(a,base)\,)(a) = \Delta(c,a)(a) = c$. Hence, $b$ is a solution to the equation $a[\![base]\!]x = c$. Also, it follows from above that $\Delta(c,a) \leq \Delta(x,base)$ for any solution $x$ of the equation $a[\![base]\!]x = c$. □

Thus, we see that whenever a compatible integrand exists (with respect to *base*, *a* and *c*), the proposed solution is itself a compatible integrand. However, as Figure 7.9 illustrates, there are situations where a compatible integrand does not exist. In this particular example, the statement "$c := 2 \times \pi \times r$" occurs in program *a* but not in program *base*. Consequently, it must occur in program $a[\![base]\!]x$ for any program $x$. Since this statement does not occur in program $c$, there does not exist any program $x$ such that $a[\![base]\!]x = c$.

| $base$ | $a$ | $c$ | $\Delta(c,a)(base)$ |
|---|---|---|---|
| **program**<br>$\pi := 3.14$<br>$r := 2$<br>$h := 3$<br>**end()** | **program**<br>$\pi := 3.14$<br>$r := 2$<br>$h := 3$<br>$c := 2 \times \pi \times r$<br>$a := \pi \times r \times r$<br>**end(**$c,a$**)** | **program**<br>$\pi := 3.14$<br>$r := 2$<br>$h := 3$<br>$a := \pi \times r \times r$<br>$v := a \times h$<br>**end(**$a,v$**)** | **program**<br>$\pi := 3.14$<br>$r := 2$<br>$h := 3$<br>$a := \pi \times r \times r$<br>$v := a \times h$<br>**end(**$v$**)** |

**Figure 7.9.** An example where a compatible integrand does not exist, and the proposed solution for separating consecutive edits.

### 7.2. Δ(c,a)(base) versus c[[a]]base

**Theorem 7.3.** If $\Delta(c,a)$ and $\Delta(base,a)$ are compatible then $\Delta(c,a)(base) = c[\![a]\!]base$.

**Proof.** Follows immediately from Proposition 6.3($a$). □

Figure 7.10 shows an example where $c[\![a]\!]base$ is different from $\Delta(c,a)$. The example illustrates that $c[\![a]\!]base$ ends up restoring elements that were discarded in going from *a* to *c*, while $\Delta(c,a)(base)$ more appropriately leaves them out. This happens because $c[\![a]\!]base$ treats *c* and *base* symmetrically and “insertions” take precedence over “deletions”.

### 7.3. Δ(c,a)(base) versus Δ(base,a)(c)

**Theorem 7.4.** $\Delta(c,a)(base) = \Delta(base,a)(c)$ iff $\Delta(c,a)$ and $\Delta(base,a)$ are compatible.

**Proof.** If $\Delta(c,a)$ and $\Delta(base,a)$ are compatible, it follows from Proposition 6.3($a$) that $\Delta(c,a)(base) = \Delta(base,a)(c)$. On the other hand, if $\Delta(c,a)$ and $\Delta(base,a)$ do conflict, it

| $base$ | $a$ | $c$ | $\Delta(c,a)(base)$ | $c[\![a]\!]base$ |
|---|---|---|---|---|
| **program**<br>$x := 0$<br>$y := x$<br>$z := y$<br>**end(**$x,y,z$**)** | **program**<br>$x := 0$<br>$y := x$<br>**end(**$x,y$**)** | **program**<br>$x := 0$<br>**end(**$x$**)** | **program**<br>$x := 0$<br>**end(**$x$**)** | **program**<br>$x := 0$<br>$y := x$<br>$z := y$<br>**end(**$x,y,z$**)** |

**Figure 7.10.** An example where $\Delta(c,a)(base)$ and $c[\![a]\!]base$ differ.

follows from Proposition 4.16 that $(\Delta(c,a) \circ \Delta(base,a))\,(a) \neq (\Delta(base,a) \circ \Delta(c,a))\,(a)$. Hence, $\Delta(c,a)(base) \neq \Delta(base,a)(c)$. □

Figure 7.11 gives an example where $\Delta(c,a)(base)$ and $\Delta(base,a)(c)$ differ. Note that $\Delta(base,a)(c)$, unlike $\Delta(c,a)(base)$, ends up inappropriately omitting the computation of $v$ through "$v := a \times h$" which was added to $a$ in deriving $c$.

## 8. Pushouts and program-integration

We now turn our attention to the category $\mathcal{C}$ that was introduced in Section 3. (See Definition 3.8.) We are interested in the relationship between pushouts in this category and the operation of integration.

We first provide some motivation for studying this relationship. We begin by looking at some generalizations of the program-integration problem. As explained in the Introduction, the two-variant program-integration problem is only one of many situations that might arise during the program-development process. A less general version of the problem is the problem of merging software extensions, addressed by Berzins [2], where there is no *base* program—merely two programs that need to be merged. A more general version of the problem is the integration of $n$ variants of a base program, instead of just two variants. The most general version of the problem is the integration of a DAG of multiple versions of a program. Here the (possibly many) roots of the DAG represent various initial versions of the program. Other versions were obtained either by making some modification to some previous version, or by merging or integrating some of the previous versions. The problem is to generate one

| *base* | *a* | *c* | $\Delta(c,a)(base)$ | $\Delta(base,a)(c)$ |
|---|---|---|---|---|
| **program**<br>  $\pi := 3.14$<br>  $r := 2$<br>  $c := 2 \times \pi \times r$<br>**end**($c$) | **program**<br>  $\pi := 3.14$<br>  $r := 2$<br>  $a := \pi \times r \times r$<br>**end**($a$) | **program**<br>  $\pi := 3.14$<br>  $r := 2$<br>  $h := 3$<br>  $a := \pi \times r \times r$<br>  $v := a \times h$<br>**end**($a,v$) | **program**<br>  $\pi := 3.14$<br>  $r := 2$<br>  $h := 3$<br>  $c := 2 \times \pi \times r$<br>  $a := \pi \times r \times r$<br>  $v := a \times h$<br>**end**($c,v$) | **program**<br>  $\pi := 3.14$<br>  $r := 2$<br>  $h := 3$<br>  $c := 2 \times \pi \times r$<br>**end**($c$) |

**Figure 7.11.** An example where $\Delta(c,a)(base)$ and $\Delta(base,a)(c)$ differ.

single version that reconciles the multiple lines of program development.

The primary motivation for studying the algebraic properties of the program-integration operator was the following question: can the more general versions of the problem described above be solved by repeated applications of the two-variant program-integration operator, and if so, does the order in which various integration steps are done matter? Thus, for instance, if we attempt to integrate three variants *a*, *b*, and *c* of program *base* using two-variant program-integration, we face the question of whether the various “reasonable” ways of doing this are all equivalent; $(a[\![base]\!]b)\ [\![base]\!]c$ and $a[\![base]\!]\ (b[\![base]\!]c)$ are just two of the possibilities. The number of possibilities explode when we attempt to integrate a more complex version DAG.

While (two-variant) integration is a ternary function, pushout is a function of three objects, say *base*, *a* and *b*, *and* two morphisms, one between *base* and *a* and one between *base* and *b*. (Since, in a category, a morphism identifies both its domain and codomain, pushout is really only a function of two morphisms with a common domain. The three objects are implicit arguments.) There is a generalization of pushouts to arbitrary *diagrams* (consisting of multiple objects and morphisms between them), namely *co-limits*. Berzins program-merging operation, for instance, is a least-upper-bound operation, and hence *is* the co-limit of two objects (called a *sum* or *co-product*) in the category that represents Scott’s partial ordering among programs. What is interesting about co-limits is that co-limits of complex diagrams can be obtained by repeated applications of co-limits of simpler diagrams (such as pushouts and co-products). Further, all reasonable ways of composing co-limits of simpler diagrams yield the same result. These results concerning co-limits, in conjunction with the questions we raised in the previous paragraphs provide an obvious motivation for studying the question: “Is integration a pushout?”

Pushouts in a category are unique only up to isomorphism. We first look at the question of when two programs (objects) in $\mathcal{C}$ are isomorphic.

**Proposition 8.1.** The category $\mathcal{C}$ is skeletal: no two distinct objects are isomorphic.

**Proof.** Let $a$ and $b$ be isomorphic objects. Thus, there exist morphisms $[a,f,b]$ and $[b,g,a]$ such that $f \circ g = I$ and $g \circ f = I$. But, $I = f \circ g$ implies that $I \geq f$. Hence, $I = f$, since $I$ is the least element of $M$. Hence, $b = f(a) = a$. □

It is worth noting at this stage that objects in the category $\mathcal{C}$ are elements of a given double Brouwerian algebra. In the double Brouwerian algebra we are interested in, the elements are downwards-closed sets of single-point slices. Such sets are not programs but program representations. Several programs might have the same set of single-point slices. Hence, we may view an object in $\mathcal{C}$ as an equivalence class of programs. Thus, $\mathcal{C}$ is the skeleton of another category in which the objects really denote individual programs, and in which two programs are isomorphic iff they have the same set of single-point slices.

**Theorem 8.2.** The pushout of diagram 8.12($a$) exists iff $m_1$ and $m_2$ commute and $m_1 \sqcap m_2 = I$, in which case the pushout is the diagram 8.12($b$). In particular, the pushout, when it exists, does yield the integrated program.

**Proof.** See the Appendix.

Thus, we see that when the pushout of Figure 8.12($a$) exists, it is closely related to the integration operation. However, in many cases the pushout may not exist, though the result of the integration operation may be well defined. For instance, as Theorem 8.2 shows, a necessary (but not sufficient) condition for the pushout to exist is that $m_1$ and $m_2$ be compatible.

Thus we get the following partial result. Consider the integration of a DAG of versions. If all relevant sub-diagrams of the DAG have a pushout, then all alternative ways of generating the integrated program using a sequence of the two-variant program integration yield the same result. If pushouts always existed in $\mathcal{C}$ then this would be a very strong result: all rea-

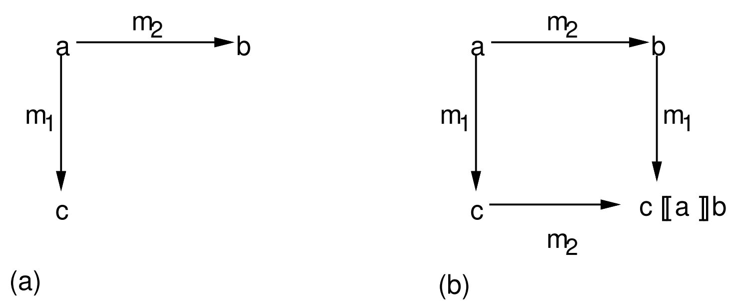


**Figure 8.12.** Is integration a pushout in $\mathcal{C}$?

sonable sequences of two-variant program integration operations on a DAG of versions yield the same result. However we will show shortly that we cannot hope to construct a category $\mathcal{D}$ in which the pushout operation and Brouwerian program-integration operation coincide. This leaves open the question of how a DAG of versions should be integrated when the pushout does not exist for all the relevant sub-diagrams, since the results obtained by using different sequences of two-variant program-integration operations can be different.

**Theorem 8.3.** Let $\mathcal{D}$ be any category whose set of objects is $P$, the set of programs. Assume that there exist at least two objects, and that there exists at least one morphism between any two objects. It is not possible that the pushout of every diagram of the form 8.13($a$) exists and equals diagram 8.13($b$). Thus, it is not possible that the pushout always exists and yields the integrated program.

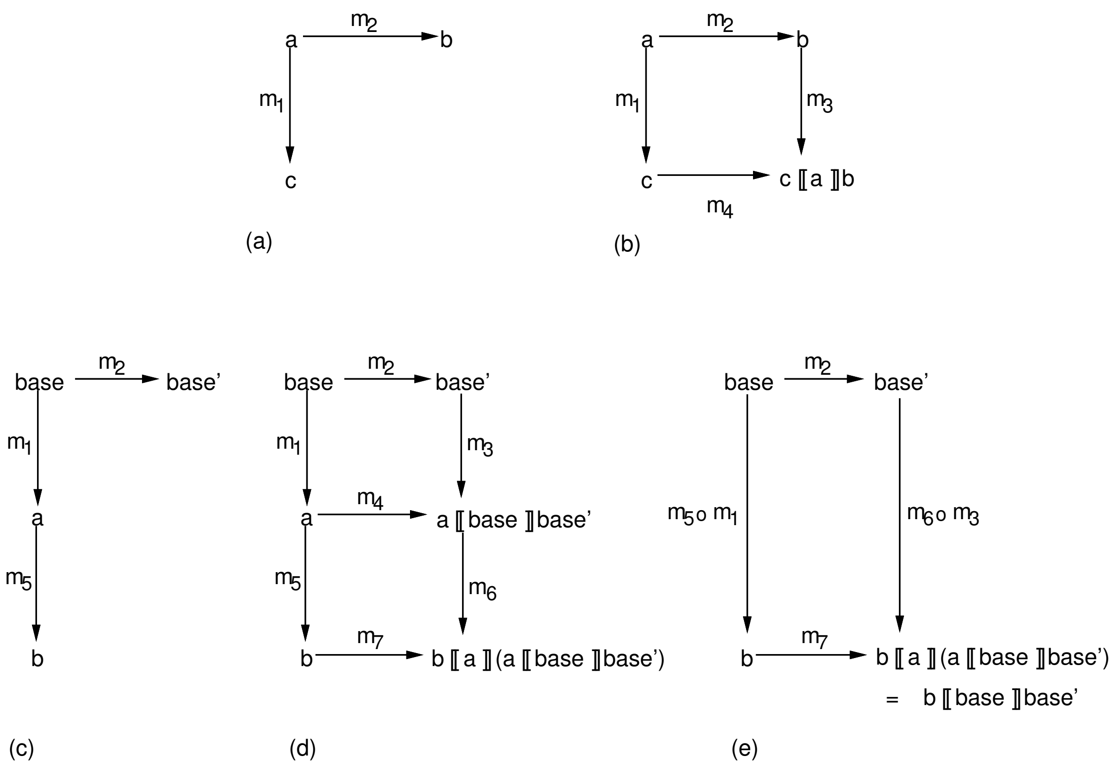


**Figure 8.13.** Is integration a pushout in some category?

**Proof.** Assume the converse. Consider the diagram 8.13($c$). The diagram is an instance of the problem of propagating changes through a tree of versions: *base*, $a$ and $b$ form a version tree, and *base′* is obtained by making some change to *base*; the problem is to propagate this change through the tree. Consider the commuting diagram 8.13($d$) obtained by first constructing $a[\![base]\!]base'$ as the pushout of $a$, *base* and *base′*, and then constructing $b[\![a]\!](a[\![base]\!]base')$ as the pushout of $b$, $a$, and $a[\![base]\!]base'$. Thus, the two inner squares are pushouts. The Pushout Lemma of category theory says that, under these conditions, the outer rectangle (diagram 8.13($e$)) must itself be a pushout. (See [5], for instance.) In particular, this means that $b[\![base]\!]base'$ must be equal to $b[\![a]\!](a[\![base]\!]base')$, for all *base*, *base′*, $a$ and $b$. But this is not true. For instance, choose *base* and $a$ to be two different programs, and choose *base′* to be $a$ and $b$ to be *base*. Reps [13] shows that the Brouwerian program integration operation satisfies the following two properties: (1) $x[\![x]\!]y = y = y[\![x]\!]x$ and (2) $y[\![x]\!]y = y$. Consequently, $b[\![base]\!]base' = base[\![base]\!]a$ (by choice of $b$ and $base'$) $= a$ (by (1)). But, $b[\![a]\!](a[\![base]\!]base') = base[\![a]\!](a[\![base]\!]a) = base[\![a]\!]a = base$. Thus, we have $a = base$, which is a contradiction. □

## Appendix

### Proof of Theorem 8.2

Recall that a diagram of the form A.14($d$) is said to be a pushout of the diagram A.14($a$) iff it commutes and for any morphisms $m_5$ and $m_6$ that make diagram A.14($e$) commute, there exists a unique morphism $m$ that makes diagram A.14($f$) commute. We will say that diagram A.14($d$) satisfies the *existence condition* if for any $m_5$ and $m_6$ that make diagram A.14($e$) commute there exists an $m$ that makes diagram A.14($f$) commute. Similarly, we will say that diagram A.14($d$) satisfies the *uniqueness condition* if for any $m_5$ and $m_6$ that make diagram A.14($e$) commute there exists at most one $m$ that makes diagram A.14($f$) commute.

$\Rightarrow$

We prove the forward implication in two steps. (1) We first show that if the pushout of diagram A.14($a$) exists, then $m_1$ and $m_2$ must commute, and the pushout must be given by the diagram A.14($c$). (2) We then show that if diagram A.14($c$) is the pushout of diagram A.14($a$) then $m_1 \sqcap m_2$ must be $I$. Note that if $m_1 \sqcap m_2 = I$ then $m_1 \dot{-} m_2 = (m_1 \dot{-} m_2) \sqcup (m_1 \sqcap m_2) = m_1$, and similarly, $m_2 \dot{-} m_1 = m_2$. Under these conditions, diagram A.14($c$)

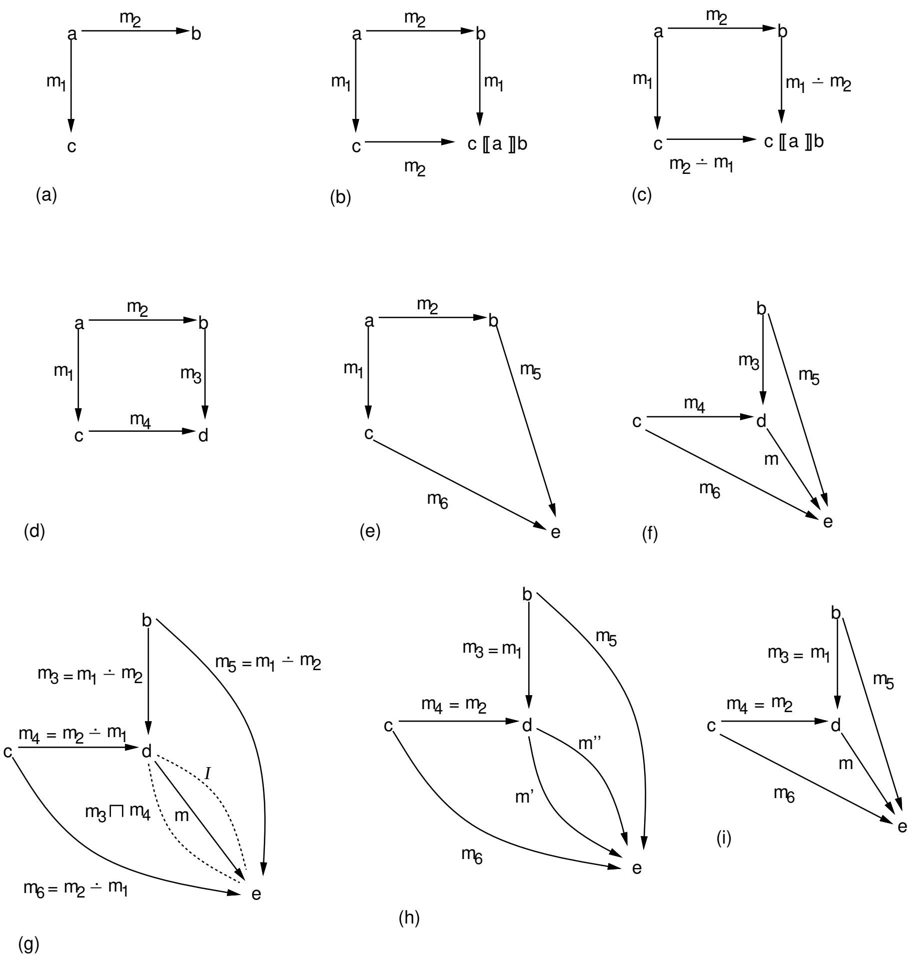


**Figure A.14.** Is integration a pushout in $\mathcal{C}$?

reduces to diagram A.14($b$).

(1) Assume that the diagram A.14($d$) is the pushout of diagram A.14($a$). Then, for any $m_5$ and $m_6$ that make diagram A.14($e$) commute, there exists a unique $m$ that makes diagram A.14($f$) commute. Thus, $m_5 = m \circ m_3 \geq m$ by Proposition 4.6($d$). Similarly, $m_6 \geq m$. Further, since $m \circ m_3 = m_5$, we get (using Proposition 4.12($d$)) $m_3 \geq m_5 \dot{-} m$.

Choose $m_5$ to be $m_1$ and $m_6$ to be $m_2 |_c m_1$ — this choice makes diagram A.14($e$) commute, as shown in Proposition 4.12($c$) and Figure 4.4. Hence, from the previous inequality, we see that $m_3 \geq m_5 \dot{-} m \geq m_5 \dot{-} m_6$ (since $m \leq m_6$) $\geq m_1 \dot{-} (m_2 |_c m_1) \geq m_1 \dot{-} m_2$. Similarly, we derive that $m_4 \geq m_2 \dot{-} m_1$.

Now, $m_3$ and $m_4$ commute, since $m_4 \circ m_1 = m_3 \circ m_2$ is an upper bound for $m_3$ and $m_4$. Hence, $m_1 \dot{-} m_2$ and $m_2 \dot{-} m_1$ commute (Proposition 4.8($b$)). This implies that $m_1$ and $m_2$ commute as follows. $m_1 \dot{-} m_2$ commutes with $m_2 \sqcap m_1$ since both have an upper bound, namely $m_1$. Since $m_1 \dot{-} m_2$ commutes with both $m_2 \dot{-} m_1$ and $m_2 \sqcap m_1$, it commutes with their composition namely $(m_2 \dot{-} m_1) \circ (m_2 \sqcap m_1)$, which is $m_2$. Similarly, since $m_2$ commutes with both $m_1 \dot{-} m_2$ and $m_1 \sqcap m_2$, it commutes with $(m_1 \dot{-} m_2) \circ (m_1 \sqcap m_2)$, which is $m_1$.

Now, choose $m_5$ to be $m_1 \dot{-} m_2$ and $m_6$ to be $m_2 \dot{-} m_1$. Since $m_1$ and $m_2$ commute, this choice makes diagram A.14($e$) commute. We thus get $m \leq m_5 = m_1 \dot{-} m_2 \leq m_3$. Thus, $m_3 = m \circ m_3 = m_5 = m_1 \dot{-} m_2$. Similarly, $m_4 = m_2 \dot{-} m_1$.

It follows that $d = (m_3 \circ m_2)(a) = ((m_1 \dot{-} m_2) \circ m_2)(a) = (m_1 \circ m_2)(a) = (m_1 \sqcup m_2)(a) = m_1(a)[\![a]\!]m_2(a) = c[\![a]\!]b$, by Proposition 6.4.

(2) Now assume that $m_1$ and $m_2$ commute, and that diagram A.14($c$) is the pushout of diagram A.14($a$). We need to show that $m_1 \sqcap m_2 = I$.

We first use the fact that A.14($c$) satisfies the uniqueness condition to establish that $(m_1 \dot{-} m_2) \sqcap (m_2 \dot{-} m_1) = I$. Choose $m_6$ to be $m_2 \dot{-} m_1$ and $m_5$ to be $m_1 \dot{-} m_2$. Since this choice makes diagram A.14($e$) commute, there must exist a unique $m$ that makes diagram A.14($g$) commute. But diagram A.14($g$) commutes if we let $m$ be either $I$ or $m_3 \sqcap m_4 = (m_1 \dot{-} m_2) \sqcap (m_2 \dot{-} m_1)$. Hence, from the uniqueness condition, $(m_1 \dot{-} m_2) \sqcap (m_2 \dot{-} m_1)$ must be equal to $I$.

Let $m_7$ denote $(m_1 \sqcap m_2) \sqcap (m_2 \dot{-} m_1)$, and let $m_8$ denote $(m_1 \sqcap m_2) \sqcap (m_1 \dot{-} m_2)$. We now use the fact that A.14($c$) satisfies the existence condition to show that $m_7 = I$ and $m_8 = I$.

$$
\begin{aligned}
& (m_1 \dot{-} m_2) \sqcap (m_2 \dot{-} m_1) = I && \text{(as shown above)} \\
\Rightarrow\ & (m_1 \dot{-} m_2) \sqcap (m_2 \dot{-} m_1) \sqcap (m_1 \sqcap m_2) = I \\
\Rightarrow\ & (m_1 \dot{-} m_2) \sqcap m_7 = I \\
\Rightarrow\ & m_3 \sqcap m_7 = I \\
\Rightarrow\ & (m_3 |_c m) \sqcap m_7 = I \quad \text{(for every } m\text{)}
\end{aligned}
$$

Assume that $m_7$ is $<x,y>$. We now show that $x = \perp$. Choose $m_5$ to be $<x,x>$ and $m_6$ to be $<\perp,x>$. This choice makes diagram A.14($e$) commute. From the commutativity of A.14($f$)

we get

$$
\begin{aligned}
& m \sqcup (m_3 \,|_c m) = m \circ m_3 = m_5 && \text{(by Proposition 4.12}(c)\text{)} \\
\Rightarrow\ & (m \sqcup (m_3 \,|_c m)) \sqcap m_7 = m_5 \sqcap m_7 = m_7 && \text{(since } m_7 = \langle x,y \rangle \le \langle x,x \rangle = m_5\text{)} \\
\Rightarrow\ & (m \sqcap m_7) \sqcup ((m_3 \,|_c m) \sqcap m_7) = m_7 \\
\Rightarrow\ & (m \sqcap m_7) \sqcup I = m_7 \\
\Rightarrow\ & m \sqcap m_7 = m_7 \\
\Rightarrow\ & m \ge m_7 \\
\Rightarrow\ & m_6 \ge m_7 && \text{(since } m_6 \ge m\text{)} \\
\Rightarrow\ & \langle \bot, x \rangle \ge \langle x,y \rangle \\
\Rightarrow\ & \bot \sqsupseteq x \\
\Rightarrow\ & x = \bot.
\end{aligned}
$$

By a similar reasoning we can show that $y = \top$. Consequently, $m_7 = I$. Just as we showed $m_7 = I$, we can show that $m_8 = I$.

We now use the fact that A.14($c$) satisfies the existence condition to show that $m_1 \sqcap m_2 \le (m_2 \dot{-} m_1) \sqcup (m_1 \dot{-} m_2)$. Choose $m_6$ to be $m_2 \dot{-} m_1$ and $m_5$ to be $m_1$. Consider diagram A.14($f$) (with $m_4 = m_2 \dot{-} m_1$ and $m_3 = m_1 \dot{-} m_2$). Since $m \circ m_4 = m_6$, we have $m \le m_6 = m_2 \dot{-} m_1$. Hence, $m$ and $m_3 = m_1 \dot{-} m_2$ commute, and we get $m_1 = m_5 = m \circ m_3 = m \sqcup m_3 = m \sqcup (m_1 \dot{-} m_2) \le (m_2 \dot{-} m_1) \sqcup (m_1 \dot{-} m_2)$. Consequently, $m_1 \sqcap m_2 \le m_1 \le (m_2 \dot{-} m_1) \sqcup (m_1 \dot{-} m_2)$.

Using this,

$$
\begin{aligned}
m_1 \sqcap m_2 &= (m_1 \sqcap m_2) \sqcap ((m_2 \dot{-} m_1) \sqcup (m_1 \dot{-} m_2)) \\
&= [(m_1 \sqcap m_2) \sqcap (m_2 \dot{-} m_1)] \sqcup [(m_1 \sqcap m_2) \sqcap (m_1 \dot{-} m_2)] \\
&= m_7 \sqcup m_8 \\
&= I
\end{aligned}
$$

$\Leftarrow$

We now assume that $m_1$ and $m_2$ commute, and that $m_1 \sqcap m_2 = I$, and show that diagram A.14($b$) is the pushout of diagram A.14($a$). Obviously, diagram A.14($b$) commutes.

We first show that if diagram A.14($b$) does not satisfy the uniqueness condition then $m_1 \sqcap m_2 > I$, contradicting our assumptions. Thus, for some $m_5$ and $m_6$ that make A.14($e$) commute there exist two different modifications $m'$ and $m''$ such that diagram A.14($h$) commutes. We then have $m'' \circ m_1 = m_5$. Hence, $m_5 \ge m''$. We also have $m' \circ m_1 = m_5$. Hence, from Proposition 4.12($d$), $m_1 \geq m_5 \dot{-} m' \geq m'' \dot{-} m'$. Similarly, $m_1 \geq m' \dot{-} m''$. Combining both inequalities, we get $m_1 \geq (m' \dot{-} m'') \sqcup (m'' \dot{-} m')$. Similarly, we get $m_2 \geq (m' \dot{-} m'') \sqcup (m'' \dot{-} m')$. Hence, $m_1 \sqcap m_2 \geq (m' \dot{-} m'') \sqcup (m'' \dot{-} m')$. But if $m'$ and $m''$ are different, then $(m' \dot{-} m'') \sqcup (m'' \dot{-} m')$ is strictly greater than $I$. (This is because if $(m' \dot{-} m'') \sqcup (m'' \dot{-} m') = I$, then $(m' \dot{-} m'') \leq I$ and $(m'' \dot{-} m') \leq I$, which implies that $m' \leq m''$ and $m'' \leq m'$, which, in turn, implies that $m' = m''$.) Hence, $m_1 \sqcap m_2 > I$, contradicting our assumption. Therefore, A.14($b$) must satisfy the uniqueness condition.

We now show that diagram A.14($b$) satisfies the existence condition. Given $m_5$ and $m_6$ such that diagram A.14($e$) commutes, we need to find an $m$ such that diagram A.14($i$) commutes. We choose $m$ to be $(m_6 \dot{-} m_2) \sqcup (m_5 \dot{-} m_1)$. Note that since $m_6 \circ m_1 = m_5 \circ m_2$ is an upper bound for both $m_6 \dot{-} m_2$ and $m_5 \dot{-} m_1$, the least upper bound of $m_6 \dot{-} m_2$ and $m_5 \dot{-} m_1$ exists. From the symmetry of the problem in $m_5$ and $m_6$, it suffices to show that $m \circ m_1 = m_5$. From the commutativity of A.14($e$), we get $m_5 \circ m_2 = m_6 \circ m_1 \geq m_6$. Hence, from Proposition 4.12($e$), $m_5 \geq m_6 \dot{-} m_2$. Now,

$$
\begin{aligned}
m \circ m_1 &= ((m_6 \dot{-} m_2) \sqcup (m_5 \dot{-} m_1)) \circ m_1 && \\
&= (m_6 \dot{-} m_2) \circ (m_5 \dot{-} m_1) \circ m_1 && \\
&= (m_6 \dot{-} m_2) \circ (m_5 \circ m_1) && \text{(by Proposition 4.12}(a)) \\
&= m_5 \circ m_1 && \text{(since } (m_6 \dot{-} m_2) \leq m_5 \leq (m_5 \circ m_1)\text{ )} \\
&= m_5 \sqcup (m_1 |_c m_5) && \text{(by Proposition 4.12}(c)) \qquad (\dagger)
\end{aligned}
$$

So it is sufficient to show that $(m_1 |_c m_5) \leq m_5$. Now $m_1 |_c m_5$ is compatible with both $m_5$ and $m_2$. Hence, $m_1 |_c m_5$ is compatible with $m_5 \circ m_2$. Since $m_6 \leq m_5 \circ m_2$ and compatibility is preserved downwards, $m_1 |_c m_5$ is compatible with $m_6$ too. Since $m_1 |_c m_5$ is compatible with $m_6$ and $m_1 |_c m_5 \leq m_1$, Proposition 4.11($c$) implies that $m_1 |_c m_5 \leq m_1 |_c m_6$. Thus,

$$
\begin{aligned}
m_5 \sqcup (m_2 |_c m_5) &= m_5 \circ m_2 \\
&= m_6 \circ m_1 \\
&= m_6 \sqcup (m_1 |_c m_6) \\
&\geq m_1 |_c m_6 \\
&\geq m_1 |_c m_5 .
\end{aligned}
$$

Hence, taking meet with $m_1$ on both sides, we get

$$
\begin{aligned}
& (m_5 \sqcap m_1) \sqcup ((m_2 |_c m_5) \sqcap m_1) \geq (m_1 |_c m_5) \sqcap m_1 \\
\Rightarrow\ & (m_5 \sqcap m_1) \sqcup (m_2 \sqcap m_1) \geq m_1 |_c m_5 \\
\Rightarrow\ & (m_5 \sqcap m_1) \sqcup I \geq m_1 |_c m_5
\end{aligned}
$$

$$\Rightarrow (m_5 \sqcap m_1) \quad \geq m_1 |_c m_5$$
$$\Rightarrow m_5 \quad \geq (m_1 |_c m_5).$$

Hence, the earlier equation (†) $m \circ m_1 = m_5 \sqcup (m_1 |_c m_5)$ simplifies to $m \circ m_1 = m_5$. This completes the proof that $m \circ m_1 = m_5$. Similarly, $m \circ m_2 = m_6$, and $m$ does make diagram A.14($f$) commute. Hence, if $m_1 \sqcap m_2$ is $I$, diagram A.14($b$) satisfies the existence condition.

Hence, A.14($b$) is the pushout of A.14($a$). □

**Algebraic Laws for Brouwerian Algebras**

This appendix covers some of the algebraic laws that hold for Brouwerian algebras and double Brouwerian algebras. The material presented here is meant to make this paper self-contained; proofs of the following propositions and further information about Brouwerian algebras can be found in [8, 11, 14]. The dual of each of the following propositions, which is obtained by substituting $\top$ for $\perp$, $\perp$ for $\top$, $\sqcup$ for $\sqcap$, $\sqcap$ for $\sqcup$, $\dot{-}$ for $\div$, $\div$ for $\dot{-}$, $\sqsubseteq$ for $\sqsupseteq$, and $\sqsupseteq$ for $\sqsubseteq$, also holds true in a double Brouwerian algebra.

| | |
|---|---|
| **A.1.** $b \dot{-} a = \perp$ *iff* $a \sqsupseteq b$. | **A.11.** $(c \dot{-} a) \sqcup (c \dot{-} b) = c \dot{-} (a \sqcap b)$. |
| **A.2.** $a \dot{-} a = \perp$. | **A.12.** $(c \dot{-} b) \dot{-} a = c \dot{-} (a \sqcup b) = (c \dot{-} a) \dot{-} b$. |
| **A.3.** $\perp \dot{-} a = \perp$. | **A.13.** $(b \dot{-} a) \sqcup (c \dot{-} b) \sqsupseteq c \dot{-} a$. |
| **A.4.** $b \dot{-} \perp = b$. | **A.14.** $a \sqsupseteq (a \sqcup b) \dot{-} b$. |
| **A.5.** $a \sqcup (b \dot{-} a) \sqsupseteq b$. | **A.15.** $(a \sqcap b) \dot{-} c \sqsubseteq a \sqcap (b \dot{-} c)$. |
| **A.6.** *If* $a_1 \sqsubseteq a_2$ *then* $b \dot{-} a_2 \sqsubseteq b \dot{-} a_1$. | **A.16.** $b \dot{-} (b \dot{-} a) \sqsubseteq a \sqcap b$. |
| **A.7.** *If* $b_1 \sqsubseteq b_2$ *then* $b_1 \dot{-} a \sqsubseteq b_2 \dot{-} a$. | **A.17.** $c \dot{-} (a \sqcup b) \sqsubseteq (c \dot{-} a) \sqcap (c \dot{-} b)$. |
| **A.8.** $b \sqsupseteq b \dot{-} a$. | **A.18.** $(a \sqcup b) \dot{-} (a \sqcap b) = (a \dot{-} b) \sqcup (b \dot{-} a)$. |
| **A.9.** $a \sqcup (b \dot{-} a) = a \sqcup b$. | **A.19.** $a \dot{-} (a \sqcap b) = a \dot{-} b$. |
| **A.10.** $(b \dot{-} a) \sqcup (c \dot{-} a) = (b \sqcup c) \dot{-} a$. | |

REFERENCES


1. Asperti, A. and Longo, G., *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist,* The MIT Press, Cambridge, MA (1991).

2. Berzins, V., "On merging software extensions," *Acta Informatica* **23** pp. 607-619 (1986).

3. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.* **9**(3) pp. 319-349 (July 1987).

4. Fraser, C.W., “Maintaining program variants by merging editor scripts,” *Software—Practice and Experience* **10** pp. 817-821 (1980).

5. Goldblatt, R., *Topoi: The Categorical Analysis of Logic, Studies in Logic and the Foundations of Mathematics,* Vol. 98, North-Holland, Amsterdam (1979, Revised Edition: 1984).

6. Horwitz, S., Prins, J., and Reps, T., “Integrating non-interfering versions of programs,” *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

7. Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., “Dependence graphs and compiler optimizations,” pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages,* (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).

8. McKinsey, J.C.C. and Tarski, A., “On closed elements in closure algebras,” *Annals of Mathematics* **47**(1) pp. 122-162 (January 1946).

9. Ramalingam, G. and Reps, T., “A theory of program modifications,” pp. 137-152 in *Proceedings of the Colloquium on Combining Paradigms for Software Development,* (Brighton, UK, April 8-12, 1991), *Lecture Notes in Computer Science,* Vol. 494, ed. S. Abramsky and T.S.E. Maibaum,Springer-Verlag, New York, NY (1991).

10. Ramalingam, G. and Reps, T., “Modification algebras,” in *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology (AMAST),* (Iowa City, Iowa, May 22-25, 1991), ed. C. Rattray and T. Rus,Springer-Verlag (1992).

11. Rasiowa, H. and Sikorski, R., *The Mathematics of Metamathematics,* Polish Scientific Publishers, Warsaw (1963).

12. Reps, T. and Yang, W., “The semantics of program slicing and program integration,” pp. 360-374 in *Proceedings of the Colloquium on Current Issues in Programming Languages,* (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science,* Vol. 352, Springer-Verlag, New York, NY (1989).

13. Reps, T., “Algebraic properties of program integration,” pp. 326-340 in *Proceedings of the Third European Symposium on Programming,* (Copenhagen, Denmark, May 15-18, 1990), *Lecture Notes in Computer Science,* Vol. 432, ed. N. Jones,Springer-Verlag, New York, NY (1990).

14. Reps, T., “Algebraic properties of program integration,” *Science of Computer Programming* **17** pp. 139-215 (1991).

15. Yang, W., Horwitz, S., and Reps, T., “A program integration algorithm that accommodates semantics-preserving transformations,” To appear in *ACM Trans. Software Engineering and Methodology*, (1992).