NEW PROGRAMS FROM OLD

by

G. Ramalingam and Thomas Reps

Computer Sciences Technical Report #1057

November 1991

New Programs From Old

G. RAMALINGAM and THOMAS REPS
University of Wisconsin–Madison

There is often a need in the program-development process to generate a new version of a program that relates to existing versions of the program in some specific way. Program-development tools that assist in the generation of the new version, or possibly even automatically generate the new version from the existing versions, will obviously be of great use in this process. The problems of *merging software extensions*, *program integration*, *separating consecutive edits*, and *propagating changes through multiple versions* are all instances of situations where such tools would be welcome.

These problems give rise to various questions concerning the relationship between these problems and between various strategies that may be used to tackle these problems. The goal of this paper is to address these questions.

Previous work has shown how programs may be treated as elements of a double Brouwerian algebra, and how program modifications may be treated as functions from the set of programs to itself. In this paper we study the algebraic properties of program modifications, and use them to establish various results concerning the problem of program integration and the problem of separating consecutive edits. We also show how a particular category can be constructed from programs and program modifications, and address the question of whether integration is a pushout in this category.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques – *programmer workbench*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance – *enhancement, restructuring, version control*; D.2.9 [**Software Engineering**]: Management – *programming teams, software configuration management*; F.4.m [**Mathematical Logic and Formal Languages**]: Miscellaneous; G.2.m [**Discrete Mathematics**]: Miscellaneous

General Terms: Theory

Additional Key Words and Phrases: Brouwerian algebra, dependence graph, program slice, program modification, program integration, separating consecutive edits

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.

An abridged 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) [Rama91].

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

At a first approximation, a program derivation is likely to be a directed graph in which nodes are programs and arcs are fundamental program derivation steps. W. L. Scherlis and D. S. Scott [Sche83]

1. Introduction

In the program-development process, there is often the need to generate a new version of a program that relates to the existing versions of the program in some specific way. Examples of program-development steps during which such a need arises include *merging software extensions*, *program integration*, *separating consecutive edits*, and *propagating changes through multiple versions*. Program-development tools that provide assistance during such steps—by aiding in the creation of the new version, or possibly even generating the new version from the existing versions automatically—would obviously be of great value.

The problem of *merging software extensions*, first addressed by Berzins [Berz86], is as follows: given two programs *a* and *b*, 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*". Berzins formally defines the problem to be that of generating a program whose semantics is the least upper bound (in the approximation ordering of Scott [Stoy77]) of the semantics of the two given programs. Such a program is called a *least common extension* of the given programs. The least common extension of two programs is not computable in general, and Berzins outlines (conservatively approximate) ways of generating a common extension of two programs written in a simple functional language.

The problem of *program integration*, first formalized by Horwitz et al. [Horw89], is a generalization of the above problem involving *three* versions of a program—an original *base* version and two independent revisions of the base version's source code. This can happen during the concurrent development of a program, for instance, when two programmers simultaneously modify different copies of the same program, each enhancing the program in some way or fixing some bug in the program. The program-integration problem is to determine if the two changes made have an undesirable semantic interaction, and if they do not, to generate a program that incorporates the enhancements or bug-fixes made by either of the programmers. Horwitz et al. formalized this problem for a simple programming language, and developed an integration algorithm—referred to hereafter as the HPR algorithm—for integrating programs in that language.

The HPR algorithm is *semantics-based*: the integration algorithm is based on the assumption that any change in the *behavior*, rather than the *text*, of a program variant is significant and must be preserved in the merged program. An integration system based on this algorithm will determine whether the variants incorporate interfering changes, and, if they do not, create an integrated program that includes all *changes* as well as all features of the base program that are *preserved* in all variants.

Another problem, which is similar to that of program integration in that it involves three versions of a program, is that of *separating consecutive edits*. Consider the case of two consecutive edits to a program *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. Yet another problem, the problem of *propagating changes through multiple versions*, is the following: a tree or dag of related versions of a program exists, and the goal is to make the same enhancement or bug-fix to all of them. This situation arises when multiple implementations of a program exist to support slightly different requirements. For instance, different versions of a program might support different operating systems. It was suggested by Horwitz et al. that the problems of separating consecutive edits and progagating changes were additional applications for a program-integration tool [Horw89]. For example, they suggested that by repeatedly invoking the program-integration tool it would be possible to propagate program changes through multiple versions of a program.

This paper studies the program-manipulation operations described above 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.") Our algebraic approach helps us to answer questions such as the following ones:

- What relations (if any) exist between the various program-manipulation problems described above?
- Is there any single tool that can be utilized to address these and other similar problems uniformly?
- As mentioned above, the program-integration tool was proposed as a general tool to tackle these various problems; however, we will see later that more than one strategy exists for tackling these problems using a program-integration tool. Are these strategies equivalent? If not, which is the "correct" strategy (if any)? What is the relationship between these alternative strategies?

This paper is a continuation of two earlier studies of the algebraic properties of the program-integration operation. One earlier work ([Reps90]) was motivated by the need to establish the algebraic properties of the HPR integration algorithm. There, the HPR integration algorithm was formalized as an operation in a *Brouwerian algebra* [McKi46], and the algebraic laws that hold in Brouwerian algebras were utilized in establishing properties of the HPR integration algorithm. Recently, the desire to formalize the notion of a *program modification*—the change made to one program to obtain another—motivated the introduction of a new algebraic structure in which integration can be formalized, called *fm-algebra* [Rama91a]. In *fm*-algebra, the notion of integration derives from the concepts of a *program modification* and an operation for *combining modifications.* Thus, while the earlier work reported in [Reps90] concerned a homogeneous algebra of *programs*, the later approach concerned a heterogeneous algebra of *programs* and *program modifications.*

The aim of this paper is to investigate further the algebraic structure of the domain of program modifications, in an attempt to explore the relationships between the program-integration problem and various related problems, and the relationships between different solutions to these problems. The paper studies a particular *fm*-algebra that is defined in terms of a double Brouwerian algebra. Consequently, the integration operation of the *fm*-algebra models the HPR integration algorithm, and the results we derive apply to the HPR integration algorithm.

We wish to point out that the problems of merging software extensions, program integration, separating consecutive edits, and propagating changes through multiple versions must be addressed at both the syntactic and semantic levels. Let P denote the domain of programs, which are syntactic objects. Let S denote a suitable semantic domain, and let $\mathcal{M}: P \rightarrow S$ be a semantic function assigning each program its meaning. Assume we want to formally specify an n-ary operation $Merge: P \times \cdots \times P \rightarrow P$ that is meant to generate a new version of a program from the n input programs according to some criterion. At the semantic level, we need to specify what the desired semantics of the program to be generated is in terms of the semantics of the programs that are input to the *Merge* operation. One way this can be done is by defining a corresponding n-ary semantic operation $SemanticMerge: S \times \cdots \times S \rightarrow S$, and by requiring that the *Merge* operation satisfy the equation $\mathcal{M}(Merge(p_1, \cdots, p_n)) = SemanticMerge(\mathcal{M}(p_1), \cdots, \mathcal{M}(p_n))$.

An advantage of the abstract approach taken in this paper (and our earlier ones) is that the results established can contribute to the understanding of the various program-manipulation operations at both the syntactic and semantic levels. Because our results are established using only algebraic identities and inequalities, our results hold for *all* structures whose elements obey a few simple axioms. For example, Berzins has considered one approach to defining the program-integration operation in a semantic domain [Berz91]. He extended the conventional semantic domain to a complete Boolean algebra, and defined a semantic integration operation as a ternary operator in this Boolean algebra. However, a Boolean algebra is also a Brouwerian algebra, and the Brouwerian integration operator defined in [Reps90] reduces to the operator defined in [Berz91] when the Brouwerian algebra under consideration is a Boolean algebra. Consequently, the results from [Reps90], [Rama91a], and those that we derive in this paper are all relevant to studying the integration operation at the semantic level.

The remainder of this paper is organized into seven sections. In Section 2 we review the frameworks of Brouwerian algebra and *fm*-algebra upon which this work is based. In Section 3, we construct an *fm*-algebra from a given double Brouwerian algebra. We also construct a category from this *fm*-algebra, a category in which the objects denote programs and the arrows denote program modifications. Many properties of the *fm*-algebra may be succinctly represented by commutative diagrams of this category. In Section 4, we define a partial ordering among program modifications that captures the notion of subsumption among modifications and study the algebraic structure of this partially ordered set. In Section 5, we relate the *fm*-algebraic operators to this partial ordering, define several (related) auxiliary operators that have a simple intuitive meaning, and study the properties of these operators. In Section 6, we utilize the results of the earlier sections in establishing various properties of the integration operator. In Section 7, we look at the problem of separating consecutive edits, and various of its solutions. In Section 8, we address the question of whether program integration is a pushout in the category defined in Section 3.

2. Previous work

2.1. The Brouwerian algebraic framework

We now briefly review the Brouwerian algebraic framework for program-integration developed in [Reps90]. 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. An integration algorithm like the HPR algorithm can decompose a program into its constituent components, compare two programs a and $base$ and determine 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 $base$. (This is not quite correct, 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 $(a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base)$.

We may describe the Brouwerian algebraic framework more formally as follows. The HPR integration algorithm makes use of a program representation called a *program dependence graph* [Kuck81, Ferr87]. 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 a *single-point slice* iff 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 $\leq$ is defined on the set G_1 as follows: if a and b are single-point slices, $b \leq a$ 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, $\leq$ denotes the relation "is-a-subslice-of".

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

$$DC(A) \triangleq \{\, b \in G_1 \mid \exists a \in A.(b \leq a)\}.$$

Thus, $DC(A)$ is the smallest downwards-closed set containing A, and hence 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.

$$UC(A) \triangleq \{\, b \in G_1 \mid \exists a \in A.(a \leq b)\}.$$

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 $\bar{\ }$ denotes the set-theoretic complement with respect to $\top$:

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

Reps [Reps90] shows that $(DCS, \cup, \cap, \dot{-}, \div, \top)$ is a double Brouwerian algebra (see below) and that the HPR integration algorithm can be represented by the ternary operator _[ ]_ defined by:

$$a\,[base]\,b \triangleq (a \dot{-} base) \cup (a \cap base \cap b) \cup b \dot{-} base).$$

This permits the use of the properties of Brouwerian algebra in proving various algebraic properties of program-integration.

Definition 2.1. A *Brouwerian algebra* [McKi46] 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 $\sqsubseteq$.

(ii) P is closed under $\dot{-}$.

(iii) 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 $\perp$.

Definition 2.2. A *double Brouwerian algebra* [McKi46] 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) P is closed under $\div$.

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

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

$$a\,[base]\,b \triangleq (a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base).$$

In this framework it is convenient to think of the elements of DCS as being programs, though they really are only representations of programs. Further, not every element of DCS represents a program. An element s of DCS is said to be *feasible* if there exists a program p such that $S_p = s$. Let $FDCS$ denote the set of all feasible elements of DCS. $FDCS$ is not closed with respect to the integration operation: it is possible that a, $base$ and b are feasible elements of DCS, while $a[base]b$ is an infeasible element. The HPR integration algorithm reports *interference* among the variants exactly under these conditions.

The reader is referred to [McKi46, Rasi63, Reps90, Reps] for a discussion of the algebraic laws that hold in Brouwerian algebras and double Brouwerian algebras.

2.2. 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 [Rama91a] and made use of in providing an alternative definition of integration. We briefly review this formalism below.

Definition 2.4. A *modification algebra* is a heterogeneous algebra $(P, M, \Delta, +, apply)$with two sets P and M and three operations Δ, $+$, and *apply*, with the following functionalities:

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

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

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

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). *diff* (with the option $-e$) implements Δ, while the editor *ed* implements *apply*. There is no direct notion of +, but file concatenation is one obvious choice. This choice, in fact, leads to an integration algorithm that is essentially *diff* 3.

We now consider a particular kind of the modification algebra, in which the modifications (elements of M) happen to be certain functions from P to P, and *apply* is just ordinary function application.

Definition 2.6. A *functional-modification algebra* (abbreviated *fm*-algebra) is an algebra $(P, M, \Delta, +)$ where $M \subseteq P \longrightarrow P$, and Δ and $+$ are operations with the following functionalities:

$$\Delta\colon P \times P \longrightarrow M$$
$$+\colon M \times M \longrightarrow M.$$

Definition 2.7. The 2-variant integration operator $\_[\![\_]\!]\_ : P \times P \times P \longrightarrow P$ of an *fm*-algebra is defined as follows:

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

In [Rama91a] two different systems of axioms for *fm*-algebra are introduced, and then used to study the algebraic properties of the $\_[\![\_]\!]\_$ operator. It is shown there how the HPR integration algorithm may be

[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 *fm*-algebra we are interested in, will be given in Section 5.1.

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

modelled as the integration operation of an appropriately defined *fm*-algebra, by making use of the Brouwerian-algebraic framework outlined in the Section 2.1. The definition of this *fm*-algebra and the development of its algebraic structure will be the subject of the following sections.

3. An fm-algebra from a double Brouwerian algebra

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

Definition 3.1.

$$M \triangleq \{ \lambda z.\ (z \sqcap y) \sqcup x \mid x,y \in P \}$$

The elements of M will be referred to as *modifications*. The symbols m and m_i will typically denote elements of M.

Let $\circ$ denote function composition. Thus, $m_1 \circ m_2$ denotes the function $\lambda x.\ m_1(m_2(x))$. Then, $\lambda z.\ (z \sqcap y) \sqcup x = (\lambda z.\ z \sqcup x) \circ (\lambda z.\ z \sqcap y)$. Informally, we may think of modification $\lambda z.\ (z \sqcap y) \sqcup x = (\lambda z.\ z \sqcup x) \circ (\lambda z.\ z \sqcap y)$ 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).

We would like to represent the modification $\lambda z.\ (z \sqcap y) \sqcup x$ by the ordered pair (x,y). However, note that $\lambda z.\ (z \sqcap y_1) \sqcup x_1$ and $\lambda x.\ (x \sqcap y_2) \sqcup x_2$ might be equal even if $(x_1,y_1) \neq (x_2,y_2)$. But the representation of a modification as an ordered pair may be made unique by restricting attention to ordered pairs (x,y) where $x \sqsubseteq y$.

Definition 3.2. S, a subset of $P \times P$, is defined as follows:

$$S \triangleq \{ (x,y) \in P \times P \mid x \sqsubseteq y \}$$

Definition 3.3. A function ρ from S to M is defined as follows:

$$\rho((x,y)) \triangleq \lambda z.\ (z \sqcap y) \sqcup x$$

Proposition 3.4. ρ is a bijection from S to M.

Proof. We first show that ρ is onto; that is, we show that for any modification $m \in M$, there exists an ordered pair $(x,y) \in S$ such that $\rho((x,y)) = m$. Modification m must be equal to $\lambda z.\ (z \sqcap y_1) \sqcup x_1$ for some x_1 and y_1, by definition of M. Then, $(x_1, y_1 \sqcup x_1) \in S$, and $\rho((x_1, y_1 \sqcup x_1)) = \lambda z.\ (z \sqcap (y_1 \sqcup x_1)) \sqcup x_1 = \lambda z.\ (z \sqcap y_1) \sqcup (z \sqcap x_1) \sqcup x_1 = \lambda z.\ (z \sqcap y_1) \sqcup x_1 = m$.

We now show that ρ is 1-1; that is, we show that if (x_1,y_1) and (x_2,y_2) are two different elements of S, then $\rho((x_1,y_1)) \neq \rho((x_2,y_2))$. Observe that if $(x,y) \in S$ then $\rho((x,y))$ identifies x and y uniquely as follows: $\rho((x,y))(\bot) = x$, and $\rho((x,y))(\top) = y$. It follows that ρ is 1-1.

Hence, ρ is a bijection, and we see that $\rho^{-1}(m) = (m(\bot), m(\top))$. □

We use the bijection ρ to represent modifications as ordered pairs. We will abbreviate $\rho((x,y))$ to $<x,y>$. In particular, any reference to a modification $<x,y>$ automatically implies that $x \sqsubseteq y$.

Proposition 3.5.

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

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

(*b*) Let I denote the modification $<\bot,\top>$. 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)

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

Hence, $<x_1,y_1> \circ <x_2,y_2> \in M$. Recall that if $m \in M$, then $m = <m(\bot), m(\top)>$. 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. $<\bot,\top>$ 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. □

An intuitive explanation of program-integration using pictures has often raised the conjecture that integration was a pushout in an appropriate category. (See Figure 1.) 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.

Definition 3.6. Define a category C as follows: The set of objects is P. For every $a \in P$, and $m \in M$, there is an arrow labelled $[a,m,m(a)]$ with domain a and codomain $m(a)$. Thus, every arrow $[a,m,m(a)]$ is associated with a program modification m. (If no confusion is likely, the arrow will just be denoted by m.) The operation $\circ$ is defined to be function composition. Thus, $[b,m_2,c] \circ [a,m_1,b] = [a, m_2 \circ m_1, c]$.

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

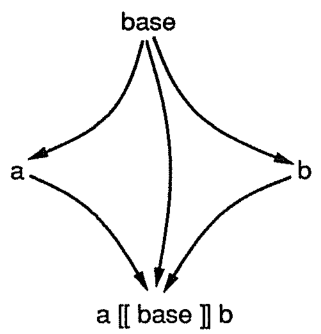


Figure 1. The problem of two variant program-integration.

4. A subsumption ordering on program-modifications

In this section we define a partial ordering $\leq$ on the set M of modifications 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 *a part of* modification m_2". This provides an intuitive meaning for the meet and join operators with respect to this partial ordering. These operators are closely linked with the function composition operator, and we begin with one of the important properties satisfied by the set M of modification functions.

Proposition 4.1. (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,

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

Corollary 4.2. $\circ$ is idempotent.

Proof. Let $m_2 = I$ in Proposition 4.1. $\square$

Definition 4.3. Define a binary relation $\leq$ on M as follows:

$m_1 \leq m_2 \iff m_1 \circ m_2 = m_2 = m_2 \circ m_1$

The relation $\leq$ is intended to capture 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. The following proposition establishes alternative interpretations of this relation.

Proposition 4.4.

(*a*) $m_1 \leq m_2$ iff $m_1 \circ m_2 = m_2$.
(*b*) $\leq$ is a partial order.
(*c*) $m_1 \leq m_2$ iff $\exists m.\ m_2 = m_1 \circ m$.
(*d*) $m_1 \circ m_2 \geq m_1$.
(*e*) I is the least element with respect to $\leq$.

Proof.

(*a*) If $m_1 \leq m_2$, then $m_1 \circ m_2 = m_2$ trivially. Conversely, assume $m_1 \circ m_2 = m_2$. Then, $m_2 \circ m_1 = (m_1 \circ m_2) \circ m_1 = m_1 \circ m_2$ (using Proposition 4.1) $= m_2$. Hence, $m_1 \leq m_2$.

(*b*) *reflexivity* : $m \circ m = m$, from Corollary 4.2. Hence, $m \leq m$, from (*a*).

antisymmetry : Assume $m_1 \leq m_2$ and $m_2 \leq m_1$. Hence, $m_1 \circ m_2 = m_2$ and $m_2 \circ m_1 = m_1$. We show that $m_1 = m_2$ as follows: $m_2 = m_1 \circ m_2 = (m_2 \circ m_1) \circ m_2 = m_2 \circ m_1$ ((using Proposition 4.1) $= m_1$.

transitivity : Assume $m_1 \leq m_2$ and $m_2 \leq m_3$. Hence, $m_1 \circ m_2 = m_2$ and $m_2 \circ m_3 = m_3$. From (*a*), we can establish that $m_1 \leq m_3$ by showing that $m_1 \circ m_3 = m_3$, which, in turn, may be established as follows: $m_1 \circ m_3 = m_1 \circ (m_2 \circ m_3) = (m_1 \circ m_2) \circ m_3 = m_2 \circ m_3 = m_3$.

(*c*) If $m_1 \leq m_2$ then $m_2 = m_1 \circ m_2$. 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, from (*a*), $m_1 \leq m_2$.

(*d*) Follows from (*c*).

(*e*) For any m, $I \circ m = m = m \circ I$. Hence, $I \leq m$. $\square$

We will often make use of Proposition 4.4(*a*), without specifically referring to it, in showing that $m_1 \leq m_2$ for some particular m_1 and m_2. The poset $(M, \leq)$ has an interesting structure, as will be apparent soon.

Consider any double Brouwerian algebra $(B, \sqcup, \sqcap, \dot{-}, \div, \top)$. Let T be any downwards-closed subset of B. Then, T will be closed with respect to the operations $\sqcap$ and $\dot{-}$, but will not necessarily be closed with respect to the operations $\sqcup$ or $\div$. $(T, \sqcup, \sqcap, \dot{-}, \div)$ is a *partial algebra*—that is, a set with a collection of partial operations—which inherits many of the algebraic properties of a double Brouwerian algebra.

Definition 4.5. A partial algebra $(T, \sqcup_1, \sqcap_1, \dot{-}_1, \div_1)$ is said to be a *partial double Brouwerian algebra* (abbreviated PDBA) if it can be extended to a double Brouwerian algebra $(B, \sqcup_2, \sqcap_2, \dot{-}_2, \div_2, \top)$ such that T is a downwards-closed subset of B, and each of the operations $\sqcup_1$, $\sqcap_1$, $\dot{-}_1$, and $\div_1$ is the restriction of the corresponding operations $\sqcup_2$, $\sqcap_2$, $\dot{-}_2$, and $\div_2$ to T.

Example. Let $(DCS, \cup, \cap, \dot{-}, \div, G)$ denote the double Brouwerian algebra of all downwards-closed sets of single-point slices. Let $FDCS$ be the set of all *feasible* slice sets $s \in DCS$. Then, $(FDCS, \cup, \cap, \dot{-}, \div)$ is a partial double Brouwerian algebra.

Note that a PDBA inherits most of the algebraic properties of double Brouwerian algebra. In particular, any identity that holds in a double Brouwerian algebra holds in a PDBA too, as long as all the relevant expressions are defined.

Definition 4.6. Let $\mathcal{B} = (B, \sqcup, \sqcap, \dot{-}, \div, \top)$ be a double Brouwerian algebra. $\overline{\mathcal{B}}$ denotes its dual $(B, \sqcap, \sqcup, \div, \dot{-}, \bot)$. $\mathcal{B} \times \overline{\mathcal{B}}$ denotes the product algebra of $\mathcal{B}$ and $\overline{\mathcal{B}}$.

Note that if $\mathcal{B}$ is any double Brouwerian algebra, then both $\overline{\mathcal{B}}$ and $\mathcal{B} \times \overline{\mathcal{B}}$ are double Brouwerian algebras. The double Brouwerian algebra $\mathcal{P} \times \overline{\mathcal{P}}$ will be of interest to us. Note that elements of this product algebra are ordered pairs of programs, and the partial ordering $\sqsubseteq_{\mathcal{P} \times \overline{\mathcal{P}}}$ on these ordered pairs is given by $(x_1,y_1) \sqsubseteq_{\mathcal{P} \times \overline{\mathcal{P}}} (x_2,y_2)$ iff $(x_1 \sqsubseteq x_2)$ and $(y_1 \sqsupseteq y_2)$. We will be particularly interested in the partial algebra of ordered pairs of programs of the form (x,y) where $x \sqsubseteq y$.

Definition 4.7. Let $\mathcal{S}$ denote the partial double Brouwerian algebra induced by S as a subset of $P \times P$ in $\mathcal{P} \times \overline{\mathcal{P}}$, where S is as defined in Definition 3.2; namely, $S = \{ (x,y) \in P \times P \mid x \sqsubseteq y \}$.

Proposition 4.8. The function ρ (Definition 3.3) is a poset isomorphism from $(M, \leq)$ to $\mathcal{S}$.

Proof. Here, $\mathcal{S}$, a partial double Brouwerian algebra, is treated as a poset. Let $m_1 = \rho((x_1,y_1))$ and $m_2 = \rho((x_2,y_2))$. Note in what follows that $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$. Now $m_1 \leq m_2$ iff $m_1 \circ m_2 = m_2$ iff $\rho((x_1,y_1)) \circ \rho((x_2,y_2)) = \rho((x_2,y_2))$ iff $\rho((x_1 \sqcup (x_2 \sqcap y_1), y_1 \sqcap (y_2 \sqcup x_1))) = \rho((x_2,y_2))$ 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 $y_1 \sqsupseteq y_2$ iff $(x_1,y_1) \sqsubseteq (x_2,y_2)$ in $\mathcal{S}$. □

It follows from the above proposition that M is itself a partial double Brouwerian algebra with respect to the partial order $\leq$. We denote the corresponding meet and (partial) join operators $\sqcap$ and $\sqcup$, and the pseudo-difference and (partial) quotient operators $\dot{-}$ and $\div$. The above proposition implies that $<x_1,y_1> \leq <x_2,y_2>$ iff $(x_1 \sqsubseteq x_2)$ and $(y_1 \sqsupseteq y_2)$. (Note that the ordering on the respective second components is the dual of the ordering on the respective first components.) Hence, for instance, $<x_1,y_1> \sqcap <x_2,y_2>$ is $<x_1 \sqcap x_2, y_1 \sqcup y_2>$. The following properties are a consequence of the fact that M is a PDBA.

Proposition 4.9.

(*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) This follows directly from the definition of a PDBA.

(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 $b \sqsupseteq (c \dot{-} a)$. □

Now, if $m_1 \leq m_2$, then we may think of modification m_1 as being a "part" of modification m_2. We may think of $m_1 \sqcap m_2$ as the modification that is "common" to the two modifications m_1 and m_2. Similarly, we may think of $m_1 \dot{-} m_2$ as the part of m_1 that is different from m_2. (However, just as in a Brouwerian algebra, $\dot{-}$ is a pseudo-difference operator.) Two modifications may or may not have a least upper bound. We first formalize a notion of *conflict* between modifications, and use that to establish certain results concerning the least upper bound of two modifications.

Figure 2 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 $c = (m_1 \circ m_2)(base)$. We will subsequently see in Proposition 5.8 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 4.10. 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 of the following propositions.

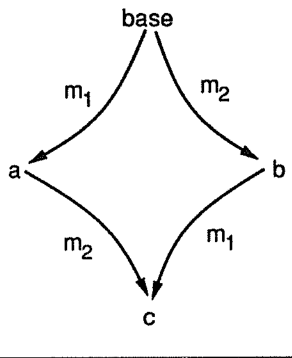


Figure 2. Integration by composing modifications.

Remark: The concept of incompatibility between modifications formalized in the above definition is distinct from the concept of *interference* in the HPR algorithm. To formalize the concept of interference, we need to work with the PDBA of feasible slice sets.

Proposition 4.11.

(*a*) m_1 and m_2 are compatible iff m_1 and m_2 have a least upper bound iff m_1 and m_2 have an 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 4.4(*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.

Since M is a PDBA, it is *consistently complete* (see Proposition 4.9(*a*)). Hence, if m_1 and m_2 have an upper bound, then they must 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 $\rho(x_1,y_1)$ and let m_2 be $\rho(x_2,y_2)$. Since ρ is a poset isomorphism between M and S, m_1 and m_2 have a least upper bound in M iff (x_1,y_1) and (x_2,y_2) have a least upper bound in S. But (x_1,y_1) and (x_2,y_2) have a least upper bound in S iff $x_1 \sqcup x_2 \sqsubseteq y_1 \sqcap y_2$. Thus, if $m_1 \sqcup m_2$ exists then $x_1 \sqcup x_2 \sqsubseteq y_1 \sqcap y_2$. Hence, $m_1 \circ m_2 = \langle x_1 \sqcup (x_2 \sqcap y_1), y_1 \sqcap (y_2 \sqcup x_1) \rangle = \langle x_1 \sqcup x_2, y_1 \sqcap y_2 \rangle = m_1 \sqcup m_2$. Similarly, $m_2 \circ m_1 = \langle x_2 \sqcup x_1, y_2 \sqcap y_1 \rangle = m_2 \sqcup m_1$. Therefore, m_1 and m_2 commute. The result follows.

(*b*) If m_1 and m_2 commute, then they have an upper bound, which is also an upper bound for m_3 and m_4. Hence, m_3 and m_4 commute. □

Let us now consider what it means for two modfications $\langle x_1,y_1 \rangle$ and $\langle x_2,y_2 \rangle$ to conflict. If the two modifications have a least upper bound, it must be $\langle x_1 \sqcup x_2, y_1 \sqcap y_2 \rangle$. Thus, the two modifications have a least upper bound iff $x_1 \sqcup x_2 \sqsubseteq y_1 \sqcap y_2$, that is, iff $x_1 \sqsubseteq y_2$ and $x_2 \sqsubseteq y_1$ (since $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$ anyway). 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.

Proposition 4.12.

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

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

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

Proof.

(*a*) Assume that $m_1 \ge m_2$. Then, $(m \circ m_2) \circ (m \circ m_1) = m \circ m_2 \circ m_1 = m \circ m_1$. Hence, $m \circ m_1 \ge m \circ m_2$ by Proposition 4.4(*a*).

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

(*c*) We have $m_1 \circ m \le m_3 \circ m$ and $m_2 \le m_3 \le m_3 \circ m$. Since $m_1 \circ m$ and m_3 have an upper bound, they commute. (Proposition 4.11(*a*)). Similarly, $m_1 \circ m$ and m_2 commute. Hence, $m_2 \circ m = (m_2 \circ m_1) \circ m = m_2 \circ (m_1 \circ m) = (m_1 \circ m) \circ m_2 \ge m_1 \circ m$. Thus we have $m_2 \circ m \ge m_1 \circ m$.

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, $m_3 \circ m = (m_3 \circ m_2) \circ m = m_3 \circ (m_2 \circ m) = (m_2 \circ m) \circ m_3 \ge m_2 \circ m$. Thus we get, $m_1 \circ m \le m_2 \circ m \le m_3 \circ m$.
□

Remark: Note that, in general, $\circ$ is not monotonic in its first argument.

5. Operations on modifications

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

5.1. Inferring modifications

The idea behind the operator Δ, 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. $\Delta(a,b)$ is supposed to represent this inferred modification. As might be expected, there is no unique modification that yields a when applied to b. We first look at the set of all such modifications. The following notation will be useful in describing such sets.

Definition 5.1. Given any modifications m_1 and m_2 such that $m_1 \le m_2$, let $[m_1,m_2]$ denote the set of modifications $\{ m \in M \mid m_1 \le m \le m_2 \}$. This is a lattice of modifications with respect to $\le$. $DC(m)$ will denote the downwards-closed set of modifications generated by m, namely $[I,m]$.

Definition 5.2. For any a and b in P, let $M_{a,b}$ denote the set of modifications that map b to a. Thus,

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

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

Proposition 5.3. $M_{a,b} = [<a \dot{-} b, a \div b>, <a,a>] = \{ m \mid <a \dot{-} b, a \div b> \le m \le <a,a> \}$. In particular, $M_{a,b}$ has a least element, namely $<a \dot{-} b, a \div b>$, and a greatest element, namely $<a,a>$.

Proof. Recall that we are interested only in the ordered pairs (x,y) satisfying $x \sqsubseteq y$. For such pairs, $(b \sqcap y) \sqcup x = (b \sqcup x) \sqcap y$.

$<x,y> \in M_{a,b} \Leftrightarrow (b \sqcap y) \sqcup x = a$ (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 $y \sqsupseteq a$

$\Leftrightarrow <x,y> \ \ge\ <a \dot{-} b, a \div b>$ and $<x,y> \ \le\ <a,a>$ □

Note that $<a,a>$, the greatest element in $M_{a,b}$, is nothing but the constant valued function $\lambda z.a$. We define $\Delta(a,b)$ conservatively, as the least modification that changes b to a, namely $<a \dot{-} b, a \div b>$. 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 6.2.)

Definition 5.4. Let $\Delta(a,b)$ denote the least element (with respect to $\le$) of $M_{a,b}$.

$$\Delta(a,b) \triangleq \min_{\le}(M_{a,b})$$

Proposition 5.5.

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

(*b*) $\Delta(m(a),a) \le m$.

(*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 Δ. (*c*) is a special case of (*b*). (*d*) follows from Proposition 5.3. As for (*e*), let a denote $m(b)$. Then, from Proposition 5.3, $\Delta(a,b) \le m \le <a,a>$. Hence, $\Delta(a,b) \dot{-} <b,b> \le m \dot{-} <b,b> \le <a,a> \dot{-} <b,b>$. That is, $\Delta(a,b) \le m \dot{-} <b,b> \le \Delta(a,b)$. Hence, the result follows. □

It can be verified easily that 5.5(a) and 5.5(b) can be used together as an equivalent axiomatic definition of Δ. We have seen that the set of modifications that take b to a, namely $M_{a,b}$, has a simple structure. We now look at some sets of modifications that are closely related to $M_{a,b}$.

Definition 5.6. For any a in P, let I_a and M_a^{-1} denote the sets of modifications defined as follows:

$$I_a \triangleq M_{a,a} = \{\, m \in M \mid m(a) = a \,\}.$$

$$M_a^{-1} \triangleq \bigcup_{b \in P} M_{b,a} = \{\, m \in M \mid \exists b \in P.(m(b) = a) \,\}$$

Note that in category C M_a^{-1} is the collection of functions associated with the collection of arrows directed to object a, and that I_a is the collection of functions associated with the set of loops on object a: arrows from a to a. In particular, modifications in I_a have no effect whatsoever on program a.

Proposition 5.7.

(a) $M_a^{-1} = I_a$.

(b) $I_a = DC(<a,a>) = \{\, <x,y> \mid x \sqsubseteq a \sqsubseteq y \,\}$.

Proof.

(a) Obviously, $M_a^{-1} \supseteq I_a$. On the other hand, let $m \in M_a^{-1}$. Then, for some program b, $m(b) = a$. Hence, $m(a) = m(m(b)) = (m \circ m)(b) = m(b) = a$. Hence, $m \in I_a$.

(b) $I_a = M_{a,a} = [<a \dot{-} a, a \div a>, <a,a>]$ (by Proposition 5.3) $= [I, <a,a>] = DC(<a,a>)$. □

Proposition 5.8. 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, both m_1 and m_2 are in the set M_b^{-1}. Hence, from Proposition 5.7, both m_1 and m_2 have an upper bound, namely $<b,b>$. Hence, from Proposition 4.11(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 4.10). 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.

5.2. Compatibility among modifications

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 5.9. Define the binary operation $|_c$ on M as follows:

$$m_1 |_c m_2 \triangleq (m_2 \circ m_1) \sqcap m_1.$$

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 $\{\, m \leq m_1 \mid P(m) \,\}$ 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 5.10.

(*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 4.4(*d*). Since $m_1 |_c m_2$ and m_2 have an upper bound (namely, $m_2 \circ m_1$) it follows from Proposition 4.11(*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 4.12(*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 commute. 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 4.11(*b*). □

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

Proposition 5.11.

(*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*) Since M is a PDBA, $m_1 = (m_1 \dot{-} m_2) \sqcup (m_1 \sqcap m_2)$ (see Proposition 4.9(*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, $(<x_1,y_1> \circ <x_2,y_2>) \dot{-} <x_2,y_2> = <x_1 \sqcup (x_2 \sqcap y_1), y_1 \sqcap (y_2 \sqcup x_1)> \dot{-} <x_2,y_2> = <x_1 \dot{-} x_2, y_1 \dot{-} y_2> = <x_1,y_1> \dot{-} <x_2,y_2>$.

(*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 4.9(*e*), $m |_c m_1 \ge m_2 \dot{-} m_1$. From Proposition 5.10(*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 4.9(*e*), $m_1 \ge m_2 \dot{-} (m |_c m_1)$. From Proposition 5.10(*b*) $m \ge m |_c m_1$, and hence, $m_2 \dot{-} (m |_c m_1) \ge m_2 \dot{-} m$. The result follows. □

$$\begin{tikzcd}
 & m_2\ (a) \arrow[dr, shift left, "m_1 \dot{-} m_2"] \arrow[dr, shift right, "m_1"'] & \\
a \arrow[ur, "m_2"] \arrow[dr, "m_1"'] & & (m_1 \circ m_2)\ (a) \\
 & m_1\ (a) \arrow[ur, "m_2 \,|_c\, m_1"'] &
\end{tikzcd}$$

Figure 3. Relation between $\circ$, $\dot{-}$ and $|_c$.

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 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 3. The fourth and fifth inequalities are generalisations of the following Brouwerian law: if $m_1 \sqcup m \geq m_2$, then $m \geq m_2 \dot{-} m_1$.

5.3. 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 earlier 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. This reasoning leads to the following definition of $+$.

Definition 5.12. $<x_1, y_1> + <x_2, y_2> \triangleq <x_1 \sqcup x_2, (y_1 \sqcap y_2) \sqcup x_1 \sqcup x_2>$.

Proposition 5.13. $(M, +, I)$ 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 $+$. □

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 5.14.

(*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*) $(m_1 |_c m_2) \sqcup (m_2 |_c m_1) \le m_1 + m_2$.

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 $\sqcup$. + itself satisfies the properties of a semilattice join operator. The corresponding partial order may be viewed as a refinement of the partial order ≤. 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.

5.4. Resolving incompatibility among modifications

Definition 5.15. Modification m_2 is said to *overrule part of* modification m_1 if $m_2 + m_1 \not\ge m_1$.

Note that the condition $m_2 + m_1 \not\ge m_1$ 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 5.16. Define the two binary operators $|_o$ and $\dot{-}_o$ as follows:

$$m_1 |_o m_2 \triangleq (m_1 + m_2) \sqcap m_1$$
$$m_1 \dot{-}_o m_2 \triangleq (m_1 |_o m_2) \dot{-} m_2$$

Proposition 5.17.

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

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

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

Proof.

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

(*b*) When we expand the definition of $|_o$ fully, we find that $<x_1, y_1> |_o <x_2, y_2> = <x_1, y_1 \sqcup x_2>$. Consequently, $(m_1 |_o m_2) |_o m_2 = m_1 |_o m_2$.

(*c*) This follows directly from the fact that $<x_1, y_1> |_o <x_2, y_2> = <x_1, y_1 \sqcup x_2>$.

(*d*) Let A denote the set $\{ m \le m_1 \mid m_2$ 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, $m \le m_1$. Hence, from (*c*), $m |_o m_2 \le m_1 |_o m_2$. But, using (*a*), $m = m |_o m_2$ since m_2 does not overrule part of m. Hence, $m \le m_1 |_o m_2$. 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 $<x_1, y_1> \le <x_2, y_2>$, 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 4.

Proposition 5.18.

(*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.

(*a*) $m_1 |_o m_2 \leq m_1$ from the definition of $|_o$. It follows from Proposition 5.14(*b*) that $m_1 |_c m_2 \leq m_1 + m_2$. Since $m_1 |_c m_2 \leq m_1$ also, it follows that $m_1 |_c m_2 \leq (m_1 + m_2) \sqcap m_1 = m_1 |_o m_2$.

(*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 Proposition 4.11(*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 4.9(*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 5.11(*a*). □

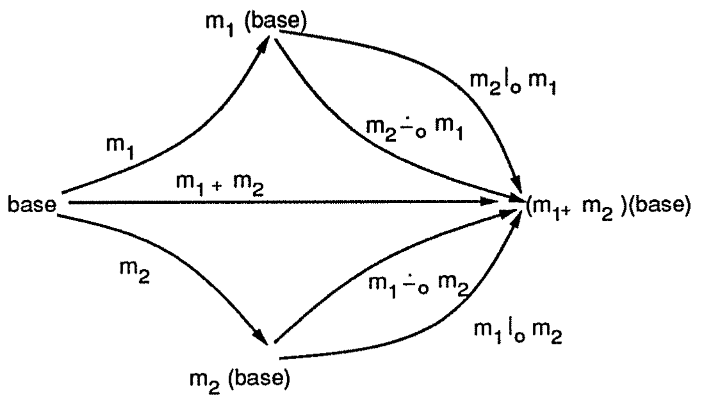


Figure 4. Relation between $|_o$, $\dot{-}_o$ and $+$.

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 is defined as follows (see Definition 2.5):

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

Proposition 6.1.

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

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

(*g*) If $\Delta(a,base)$ and $\Delta(b,base)$ are compatible then $\Delta(a[\![base]\!]b, base) = \Delta(a,base) + \Delta(b,base)$.

(*h*) $\Delta(a[\![base]\!]b, a) \le \Delta(b,base) \dot{-}_o \Delta(a,base)$.

(*i*) $\Delta(a[\![base]\!]b, a) = (\Delta(b,base) \mid_o \Delta(a,base)) \dot{-} \langle a,a \rangle$.

Proof.

(*a*)-(*e*) These follow immediately by rewriting $\Delta(a,base) + \Delta(b,base)$ in the definition of $\_[\![\_]\!]\_$ using Proposition 5.18. Thus, (*a*) follows from Proposition 5.18(*c*). (*b*) follows from Proposition 5.18(*e*). (*c*) follows from Proposition 5.18(*f*). (*d*) follows from Proposition 5.18(*g*). (*e*) follows as a special case of (*c*) since if $\Delta(a,base)$ and $\Delta(b,base)$ are compatible then $\Delta(a,base) \mid_o \Delta(b,base) = \Delta(a,base)$ from Proposition 5.18(*a*).

(*f*) This follows from the definition of $a[\![base]\!]b$ and Proposition 5.5(*b*).

(*g*) It follows from Proposition 5.5(*e*) that $\Delta(a[\![base]\!]b, base) = (\Delta(a,base) + \Delta(b,base)) \dot{-} \langle base, base \rangle$. If $\Delta(a,base)$ and $\Delta(b,base)$ are compatible, then $\Delta(a,base) + \Delta(b,base) = \Delta(a,base) \sqcup \Delta(b,base) = \langle a \dot{-} base, a \div base \rangle \sqcup \langle b \dot{-} base, b \div base \rangle = \langle (a \dot{-} base) \sqcup (b \dot{-} base), (a \div base) \sqcap (b \div base) \rangle = \langle (a \sqcup b) \dot{-} base, (a \sqcap b) \div base \rangle$. Hence, $(\Delta(a,base) + \Delta(b,base)) \dot{-} \langle base, base \rangle$ simplifies to $\Delta(a,base) + \Delta(b,base)$. The result follows.

(*h*) This follows from (*d*) and Proposition 5.5(*b*).

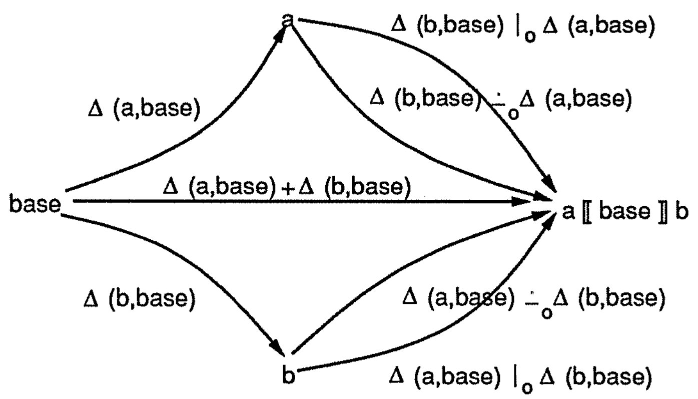
(*i*) This follows from (*c*) and Proposition 5.5(*e*). □

As seen from the proof, most of the above properties are a consequence of Proposition 5.18. Some of these results are summarized by the commutative diagram in Figure 5, which is a special case of Figure 4.

Proposition 6.2. 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 6.1(*e*). □

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



a
Δ (b,base) $|_o$ Δ (a,base)
Δ (a,base)
Δ (b,base) $\dot{-}_o$ Δ (a,base)
base
Δ (a,base) + Δ (b,base)
a ⟦ base ⟧ b
Δ (b,base)
Δ (a,base) $\dot{-}_o$ Δ (b,base)
b
Δ (a,base) $|_o$ Δ (b,base)

Figure 5. Relation between $|_o$, $\dot{-}_o$, and _⟦_⟧_.

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 [Horw89]: here, $c[\![a]\!]base$ is taken to be the desired program. A second solution previously proposed [Reps90] was that the program we seek is a solution x to the equation $a[\![base]\!]x = c$. Here, the problem is viewed as that of solving the equation $a[\![base]\!]x = c$ for x, called a "compatible integrand" with respect to *base*, *a* and *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 6. Our goal in this Section is to examine the relationship between these solutions.

Theorem 7.1. If $\Delta(c,a)$ and $\Delta(base,a)$ do not conflict then $\Delta(c,a)(base) = c[\![a]\!]base = \Delta(base,a)(c)$. However, if $\Delta(c,a)$ and $\Delta(base,a)$ do conflict, then $\Delta(c,a)(base)$ and $\Delta(base,a)(c)$ will be different, and nothing can be said about $c[\![a]\!]base$ in general.

Proof. If $\Delta(c,a)$ and $\Delta(base,a)$ do not conflict, it follows from Proposition 6.1(e) that $\Delta(c,a)(base) = c[\![a]\!]base = \Delta(base,a)(c)$. On the other hand, if $\Delta(c,a)$ and $\Delta(base,a)$ do conflict, it follows from Proposition 5.8 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 gives an example where $\Delta(c,a)(base)$ and $\Delta(base,a)(c)$ differ. In this case the re-rooting solution $c[\![a]\!]base$ coincides with the proposed solution $\Delta(c,a)(base)$, but this need not be the case in general.

Theorem 7.2. If $\Delta(c,a)$ and $\Delta(a,base)$ do not conflict then the equation $a[\![base]\!]x = c$ has a solution for x, namely $\Delta(c,a)(base)$.

Proof. We show that $b = \Delta(c,a)(base)$ satisfies the given equation. Now, $\Delta(b,base) \leq \Delta(c,a)$ (Proposition 5.5(b)). Since $\Delta(c,a)$ and $\Delta(a,base)$ are compatible, it follows that $\Delta(b,base)$ and $\Delta(a,base)$ are

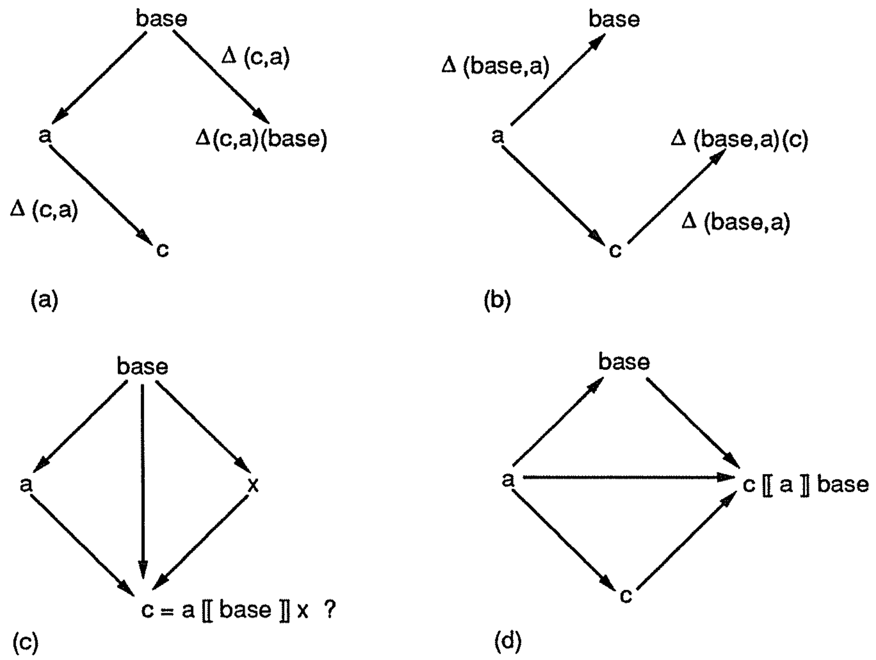


Figure 6. Different solutions to the problem of separating consecutive edits

base	*a*	*c*	$\Delta(c,a)(base) =$ $c\llbracket a\rrbracket base$	$\Delta(base,a)(c)$
program	**program**	**program**	**program**	**program**
$\pi := 3.14$	$\pi := 3.14$	$\pi := 3.14$	$\pi := 3.14$	$\pi := 3.14$
$r := 2$	$r := 2$	$r := 2$	$r := 2$	$r := 2$
$c := 2 \times \pi \times r$	$a := \pi \times r \times r$	$h := 3$	$h := 3$	$h := 3$
end(c)	**end**(a)	$a := \pi \times r \times r$	$c := 2 \times \pi \times r$	$c := 2 \times \pi \times r$
		$v := a \times h$	$a := \pi \times r \times r$	**end**(c)
		end(a,v)	$v := a \times h$	
			end(c,v)	

Figure 7. An example where $\Delta(c,a)(base)$ and $\Delta(base,a)(c)$ differ.

also compatible (Proposition 4.11(b)). Hence, from Proposition 6.1(e), we get $a\llbracket base\rrbracket b = \Delta(a,base)(b) = \Delta(a,base)\,(\Delta(c,a)(base)) = \Delta(c,a)\,(\Delta(a,base)(base)) = c$. □.

Theorem 7.3. If the equation $a\llbracket base\rrbracket x = c$ has a solution for x then $b = \Delta(c,a)(base)$ is a solution of that equation. Further, $\Delta(c,a)$ 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 $m \geq \Delta(c,a) = \Delta(b,base)$. (Or, equivalently, for any solution x of the equation, $\Delta(x,base) \geq \Delta(b,base) = \Delta(c,a)$.)

Proof. Let x be such that $a\llbracket base\rrbracket x = c$. 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 5.5($e$), and $\Delta(c,a) = (\Delta(x,base)\,|_o\,\Delta(a,base)) \dot{-} <a,a>$, from Proposition 6.1($i$). Thus, we get:

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

$$\begin{aligned}
\Delta(b,base) &= \Delta(c,a) \dot{-} <base,base> \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div a> \;\dot{-}\; <base,base> \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div a \div base> \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div (a \sqcap base)> \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div [(a \sqcap base) \sqcup (a \dot{-} base)]> \\
&\quad \text{(making use of the Brouwerian identity } (y_1 \sqcup y_2) \div z = (y_1 \sqcup y_2) \div (z \sqcup y_2)) \\
&= <(x \dot{-} a) \dot{-} base, [(x \div base) \sqcup (a \dot{-} base)] \div a> \\
&= \Delta(c,a)
\end{aligned}$$

Now, $\Delta(c,a) \leq \Delta(x,base)\,|_o\,\Delta(a,base)$ (Proposition 6.1(h)). Furthermore, $\Delta(x,base)\,|_o\,\Delta(a,base)$ is not overruled by $\Delta(a,base)$ (see Proposition 5.17b). Hence, $\Delta(c,a)$ is not overruled by $\Delta(a,base)$ (see Proposition 5.17e). Thus, $\Delta(c,a)\,|_o\,\Delta(a,base) = \Delta(c,a)$. Then, from Proposition 6.1(c), we get $a\llbracket base\rrbracket b = (\,\Delta(b,base)\,|_o\,\Delta(a,base)\,)(a) = (\,\Delta(c,a)\,|_o\,\Delta(a,base)\,)(a) = \Delta(c,a)(a) = c$. Hence, b is a solution to the equation $a\llbracket base\rrbracket x = c$. Also, it follows from above that $\Delta(c,a) \leq \Delta(x,base)$ for any solution x of the equation $a\llbracket base\rrbracket x = c$. □

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 8 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 $a[\![base]\!]x = c$.

8. Pushouts and program-integration

We now turn our attention to the category C that was introduced in Section 3. (See Definition 3.6.) 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 [Berz86], 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 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.

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

Figure 8. An example where a compatible integrand does not exist, and the proposed solution for separating consecutive edits.

While (two-variant) integration is a ternary function, pushout is a function of three objects, say *base*, *a* and *b*, *and* two arrows, one between *base* and *a* and one between *base* and *b*. (Since, in a category, an arrow identifies both its domain and codomain, pushout is really only a function of two arrows with a common domain. The three objects are implicit arguments.) There is a generalisation of pushouts to arbitrary *diagrams* (consisting of multiple objects and arrows 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). 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 C are isomorphic.

Proposition 8.1. The category C is skeletal: no two distinct objects are isomorphic.

Proof. Let a and b be isomorphic objects. Thus, there exist arrows $[a,f,b]$ 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 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 C as an equivalence class of programs. Thus, 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 9(a) exists iff m_1 and m_2 commute and $m_1 \sqcap m_2 = I$, in which case the pushout is the diagram 9(b). In particular, the pushout, when it exists, does yield the integrated program.

Proof. Recall that a diagram of the form 9(d) is said to be a pushout of the diagram 9(a) iff it commutes and for any arrows m_5 and m_6 that make diagram 9(e) commute, there exists a unique arrow m that makes diagram 9(f) commute. We will say that diagram 9(d) satisfies the *existence condition* if for any m_5 and m_6 that make diagram 9(e) commute there exists an m that makes diagram 9(f) commute. Similarly, we will say that diagram 9(d) satisfies the *uniqueness condition* if for any m_5 and m_6 that make diagram 9(e) commute there exists at most one m that makes diagram 9(f) commute.

$\Rightarrow$

We prove the forward implication in two steps. (1) We first show that if the pushout of diagram 9(a) exists, then m_1 and m_2 must commute, and the pushout must be given by the diagram 9(c). (2) We then show that if diagram 9(c) is the pushout of diagram 9(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 9(c) reduces to diagram 9(b).

(1) Assume that the diagram 9(d) is the pushout of diagram 9(a). Then, for any m_5 and m_6 that make diagram 9(e) commute, there exists a unique m that makes diagram 9(f) commute. Thus, $m_5 = m \circ m_3 \geq m$ by Proposition 4.4(d). Similarly, $m_6 \geq m$. Further, since $m \circ m_3 = m_5$, we get (using Proposition 5.11(d))

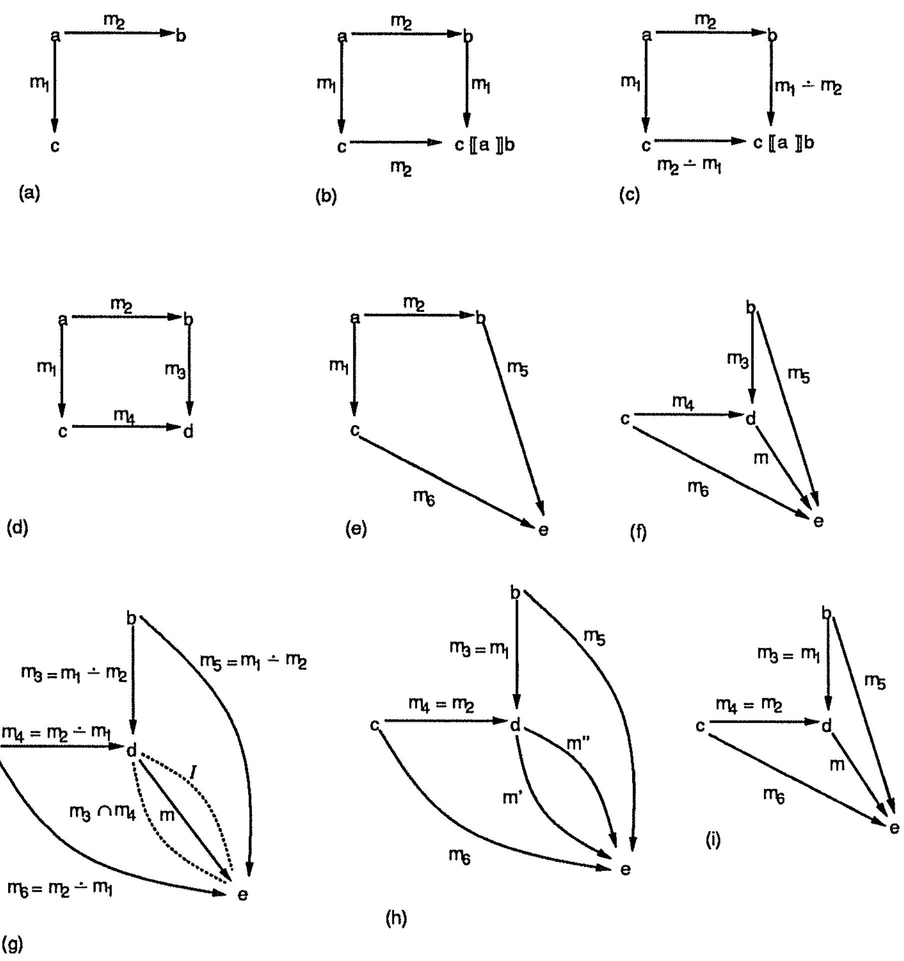


Figure 9. Is integration a pushout in C?

$m_3 \geq m_5 \dot{-} m$.

Choose m_5 to be m_1 and m_6 to be $m_2 |_c m_1$ — this choice makes diagram 9(e) commute, as shown in Proposition 5.11(c) and Figure 3. 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 4.11(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 9(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 6.2.

(2) Now assume that m_1 and m_2 commute, and that diagram 9(c) is the pushout of diagram 9(a). We need to show that $m_1 \sqcap m_2 = I$.

We first use the fact that 9(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 9(e) commute, there must exist a unique m that makes diagram 9(g) commute. But diagram 9(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 9(c) satisfies the existence condition to show that $m_7 = I$ and $m_8 = I$.

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

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 9(e) commute. From the commutativity of 9(f) we get

$$\begin{array}{ll}
m \sqcup (m_3 |_c m) = m_5 & \text{(by Proposition 5.11(}c\text{))} \\
\Rightarrow (m \sqcup (m_3 |_c m)) \sqcap m_7 = m_5 \sqcap m_7 = m_7 & \text{(since } m_7 = <x,y> \ \le \ <x,x> = m_5\text{)} \\
\Rightarrow (m \sqcap m_7) \sqcup ((m_3 |_c m) \sqcap m_7) = m_7 & \\
\Rightarrow (m \sqcap m_7) \sqcup I = m_7 & \\
\Rightarrow m \sqcap m_7 = m_7 & \\
\Rightarrow m \ge m_7 & \\
\Rightarrow m_6 \ge m_7 & \text{(since } m_6 \ge m\text{)} \\
\Rightarrow <\perp,x> \ge <x,y> & \\
\Rightarrow \perp \sqsupseteq x & \\
\Rightarrow x = \perp. &
\end{array}$$

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 9(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 9(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 9(b) is the pushout of diagram 9(a). Obviously, diagram 9(b) commutes.

We first show that if diagram 9(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 9(e) commute there exist two different modifications m' and m'' such that diagram 9(h) commutes. We then have $m'' \circ m_1 = m_5$. Hence, $m_5 \geq m''$. We also have $m' \circ m_1 = m_5$. Hence, from Proposition 5.11(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. Hence, $m_1 \sqcap m_2 > I$, contradicting our assumption. Therefore, 9(b) must satisfy the uniqueness condition.

We now show that diagram 9(b) satisfies the existence condition. Given m_5 and m_6 such that diagram 9(e) commutes, we need to find an m such that diagram 9(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 9(e), we get $m_5 \circ m_2 = m_6 \circ m_1 \geq m_6$. Hence, from Proposition 5.11(e), $m_5 \geq m_6 \dot{-} m_2$. Now,

$$\begin{aligned}
m \circ m_1 &= ((m_6 \dot{-} m_2) \sqcup (m_5 \dot{-} m_1)) \circ m_1 \\
&= (m_6 \dot{-} m_2) \circ (m_5 \dot{-} m_1) \circ m_1 \\
&= (m_6 \dot{-} m_2) \circ (m_5 \circ m_1) && \text{(by Proposition 5.11}(a)) \\
&= m_5 \circ m_1 && \text{(since } (m_6 \dot{-} m_2) \leq m_5 \leq (m_5 \circ m_1)\text{)} \\
&= m_5 \sqcup (m_1 \mid_c m_5) && \text{(by Proposition 5.11}(c)). \qquad (\dagger)
\end{aligned}$$

So it is sufficient to show that $(m_1 \mid_c m_5) \leq m_5$. Now $m_1 \mid_c m_5$ is compatible with both m_5 and m_2. Hence, $m_1 \mid_c m_5$ is compatible with $m_5 \circ m_2$. Since $m_6 \leq m_5 \circ m_2$ and compatibility is preserved downwards, $m_1 \mid_c m_5$ is compatible with m_6 too. Since $m_1 \mid_c m_5$ is compatible with m_6 and $m_1 \mid_c m_5 \leq m_1$, Proposition 5.10(c) implies that $m_1 \mid_c m_5 \leq m_1 \mid_c m_6$. Thus,

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

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

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

Hence, the earlier equation ($\dagger$) $m \circ m_1 = m_5 \sqcup (m_1 \mid_c m_5)$ 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 9(f) commute. Hence, if $m_1 \sqcap m_2$ is I, diagram 9(b) satisfies the existence condition.

Hence, 9(b) is the pushout of 9(a). □

Thus, we see that when the pushout of Figure 9(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 reasonable sequences of two-variant program integration operations on a DAG of versions yield the same result. This idea may be utilized as follows to show that we cannot hope to construct a category $\mathcal{D}$ in which the pushout operation and Brouwerian program-integration operation coincide.

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 arrow between any two objects. It is not possible that the pushout of every diagram of the form 10(*a*) exists and equals diagram 10(*b*). Thus, it is not possible that the pushout always exists and yields the integrated program.

Proof. Assume the converse. Consider the diagram 10(*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 10(*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 10(*e*)) must itself be a pushout. (See [Gold84], 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*.

$$\begin{tikzcd}
a \arrow[r, "m_2"] \arrow[d, "m_1"'] & b \\
c &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
a \arrow[r, "m_2"] \arrow[d, "m_1"'] & b \arrow[d, "m_3"] \\
c \arrow[r, "m_4"'] & c[\![a]\!]b
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
base \arrow[r, "m_2"] \arrow[d, "m_1"'] & base' \\
a \arrow[d, "m_5"'] & \\
b &
\end{tikzcd}$$

(c)

$$\begin{tikzcd}
base \arrow[r, "m_2"] \arrow[d, "m_1"'] & base' \arrow[d, "m_3"] \\
a \arrow[r, "m_4"] \arrow[d, "m_5"'] & a[\![base]\!]base' \arrow[d, "m_6"] \\
b \arrow[r, "m_7"] & b[\![a]\!](a[\![base]\!]base')
\end{tikzcd}$$

(d)

$$\begin{tikzcd}
base \arrow[r, "m_2"] \arrow[d, "m_5 \circ m_1"'] & base' \arrow[d, "m_6 \circ m_3"] \\
b \arrow[r, "m_7"] & b[\![a]\!](a[\![base]\!]base') \\
& = b[\![base]\!]base'
\end{tikzcd}$$

(e)

Figure 10. Is integration a pushout in some category?

Reps [Reps90] 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. □

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.

REFERENCES

Berz86.
Berzins, V., "On merging software extensions," *Acta Informatica* **23** pp. 607-619 (1986).

Berz91.
Berzins, V., "Software merge: models and methods for combining changes to programs," *Journal of Systems Integration* **1** pp. 121-141 (1991).

Ferr87.
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).

Gold84.
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).

Horw89.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems* **11**(3) pp. 345-387 (July 1989).

Kuck81.
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).

McKi46.
McKinsey, J.C.C. and Tarski, A., "On closed elements in closure algebras," *Annals of Mathematics* **47**(1) pp. 122-162 (January 1946).

Rama91.
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), (1991).

Rama91a.
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).

Rasi63.
Rasiowa, H. and Sikorski, R., *The Mathematics of Metamathematics,* Polish Scientific Publishers, Warsaw (1963).

Reps90.
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).

Reps.
Reps, T., "Algebraic properties of program integration," To appear in *Science of Computer Programming,* ().

Sche83.
Scherlis, W.L. and Scott, D.S., "First steps towards inferential programming," Technical report CMU-CS-83-142, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA (July 1983).

Stoy77.
Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* The M.I.T. Press, Cambridge, MA (1977).