A THEORY OF PROGRAM MODIFICATIONS

by

G. Ramalingam and Thomas Reps

Computer Sciences Technical Report #974

October 1990

A Theory of Program Modifications

G. Ramalingam and Thomas Reps
University of Wisconsin – Madison

Abstract

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to merge programs by hand. The program-integration algorithm proposed by Horwitz, Prins, and Reps provides a way to create a *semantics-based* tool for integrating a base program with two or more variants. The integration algorithm is based on the assumption that any change in the *behaviour*, 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.

This paper studies the algebraic properties of the program-integration operation, such as whether there is a law of associativity. (For example, 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.") Whereas an earlier work that studied the algebraic properties of program integration formalized the Horwitz-Prins-Reps integration algorithm as an operation in a *Brouwerian algebra*, this paper introduces a new algebraic structure in which integration can be formalized, called *fm-algebra*. In *fm*-algebra, the notion of integration derives from the concepts of a *program modification* and an operation for *combining modifications*. (Thus, while earlier work concerned an algebra of *programs*, this paper concerns an algebra of *program modifications*.) A novelty of *fm*-algebra is that program modifications are formalized as functions—more specifically, as a set of functions from programs to programs that obey a particular set of axioms.

The potential benefits of an algebraic theory of integration, such as the one developed in this paper, are actually three-fold:

(1) It allows one to understand the fundamental algebraic properties of integration—laws that express the "essence of integration." Such laws allow one to reason formally about the integration operation.
(2) It provides knowledge that is useful for designing alternative integration algorithms whose power and scope are beyond the capabilities of current algorithms.
(3) Because such a theory formalizes certain operations that are more primitive than the integration operation, an implementation of these primitive operations can form the basis for a more powerful program-manipulation system than one based on just the integration operation.

1. Introduction

The need to integrate several versions of a program into a common one arises frequently: when a system is "customized" by a user and simultaneously upgraded by a maintainer, and the user desires a customized, upgraded version; when a system is being developed by multiple programmers who may simultaneously work with separate copies of the source files; when several versions of a program exist and the same

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

Authors' address: Computer Sciences Department, University of Wisconsin – Madison, 1210 W. Dayton St., Madison, WI 53706.

enhancement or bug-fix is to be made to all of them. A tool that provides automatic assistance in tackling such problems would obviously be useful.

The program-integration algorithm proposed by Horwitz, Prins, and Reps [Horwitz89]—referred to hereafter as the HPR algorithm—provides a way to create a *semantics-based* tool for integrating two or more variants of a base program. The HPR algorithm is based on the assumption that any change in the *behaviour*, rather than the *text*, of program components in a variant is significant and must be preserved in the merged program. By the "behaviour" of a program component on some initial state σ, we mean the sequence of values produced at the component when the program is executed on σ. By the "sequence of values produced at a component," we mean: for a predicate, the sequence of boolean values to which the predicate evaluates; for an assignment statement, the sequence of values assigned to the target variable.

Given variants *A* and *B* of base program *Base*, the HPR algorithm first determines whether the changes made to *Base* to produce *A* and *B* interfere; for example, one condition that causes interference is when there is a component of *Base*, *A*, and *B* that has different behaviours in the three different programs. If there is no interference the algorithm produces a merged program *M* that incorporates the changed behaviour of *A* with respect to *Base*, the changed behaviour of *B* with respect to *Base*, and the unchanged behaviour common to *Base*, *A*, and *B*. To achieve this, the HPR algorithm employs a program representation that is similar to the *program dependence graphs* that have been used previously in vectorizing and parallelizing compilers [Kuck81, Ferrante87]. The algorithm also makes use of Weiser's notion of a *program slice* [Weiser84, Ottenstein84] to find the statements of a program that determine the behaviour of potentially affected program components.

One of the main features of the HPR algorithm that distinguishes it from text-based integration tools, such as the UNIX[1] utility *diff3*, is its semantic property described above. One has no guarantees about the way the program that results from a purely *textual* merge behaves in relation to the behaviour of the programs that are the arguments to the merge. The HPR algorithm, on the other hand, provides exactly such a semantic guarantee. This obviates the need to check the integrated program for conflicts that might have been introduced by the integration algorithm.

Though not as apparent, the algebraic properties of an integration algorithm play an equally important role in justifying various of its uses, particularly those that involve compositions of integrations. A two-variant program-integration algorithm defines a ternary (partial) function, denoted by _[_]_, on the set of programs. (*A* [*Base*]*B* denotes the result of integrating variants *A* and *B* with respect to program *Base*.) An obvious use of _[_]_ is in integrating three variants, say *A*, *B*, and *C*, of a program *Base*, by a pair of two-variant integrations. This may be done in various ways; for instance, (*A* [*Base*]*B*)[*Base*]*C* and *A* [*Base*](*B* [*Base*]*C*) are two of the possible solutions. This raises the obvious question: is _[_]_ "associative"? That is, if three variants of a given base program are to be integrated by a pair of two-variant integrations, does it matter which two variants are integrated first? Other questions about the algebraic properties of integration arise similarly from various other applications of integration.

Reps[Reps90] addressed a variety of such questions about the HPR algorithm by reformulating the HPR algorithm as an operation in a Brouwerian algebra constructed from sets of dependence graphs. (A

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

Brouwerian algebra is a distributive lattice with an additional binary operation, denoted by $\div$, which is a kind of difference operation [McKinsey46]). In this algebra, the program-integration operation can be defined solely in terms of $\sqcup$, $\sqcap$, and $\div$. By making use of the rich set of algebraic laws that hold in Brouwerian algebras, a number of algebraic properties of the integration operation were established. For instance, it was possible to show that the integration operation is associative.

One of the advantages of such an abstract approach is its potential applicability to other integration algorithms. Thus, for instance, showing that a given algorithm can be formulated as an integration operation in some Brouwerian algebra is sufficient to show that the properties proved in [Reps90] hold for the given algorithm too. Such abstract approaches are necessitated by the development of extensions and modifications to the basic HPR algorithm (intended to extend the set of language constructs to which the integration algorithms apply [Horwitz90, Horwitz89a] and to incorporate some alternative techniques [Yang90]). Unfortunately, the integration algorithm proposed in [Yang90] is not associative (and is thus not covered by the Brouwerian-algebraic framework).

This brings out another potential benefit of an algebraic theory of integration. Some properties of integration, such as associativity, are so important that they must be considered as an algebraic criterion the integration algorithms are required to satisfy. An algebraic theory of integration should not be considered as merely a tool to analyse proposed integration algorithms to discover their algebraic properties. Rather, it should provide knowledge that is of use in the design of new integration algorithms, knowledge that can ensure that these algorithms have certain algebraic properties.

Yet another of the benefits of a theory like the one developed in this paper is that it formalizes certain operators that are more primitive than the integration operation. These operators can form the basis for a more powerful program-manipulation system than one based on just the integration operation.

In this paper, we present a new approach to studying program-integration algorithms. In particular, we introduce a new algebraic structure, *fm-algebra*. Here, the concept of program integration derives from the concept of *program modifications* and the idea of *combining* program modifications. If each variant is thought of as having been obtained by performing a certain modification to the base program, an integration algorithm creates a merged program by first combining all the modifications and then applying the resultant modification to the base program. Thus, while the work reported in [Reps90] is based on an algebra of *programs*, the work reported here is based on an algebra of *program modifications*.

These ideas are formalized in Section 2, by treating "program modifications" as functional elements, by introducing an operator that combines two modifications, and by defining the integration operator in terms of these more primitive concepts. This provides us with a framework for studying the algebraic properties of integration. Section 3 lists some simple properties of the basic operators as axioms. In Section 4, some properties of the integration operator are derived from the axioms satisfied by the basic operators. Section 5 shows that this approach to integration is very general. More specifically, it is shown there that any integration operator satisfying some simple properties can be constructed in the suggested fashion from appropriately defined "program-modification" elements and a "modification-combination" operator that satisfies the proposed axioms. The construction of models of the axioms also demonstrates their consistency. We illustrate our approach with an example in Section 6. Section 7 discusses a problem related to program integration, that of separating consecutive edits on some base program into individual edits on the base program. Section 8 compares our approach with related work.

2. Functional-modification algebras

Our theory of program modifications is formalized through the concept of a functional-modification algebra, which is defined as follows:

Definition. A *functional-modification algebra* (abbreviated *fm*-algebra) is an algebra with two sets of elements, P and M, where M is a subset of P $\longrightarrow$ P. There are three operations: Δ, $*$, and function application, where Δ and $*$ have the following functionalities:

$$\Delta : P \times P \longrightarrow M$$
$$* : M \times M \longrightarrow M.$$

In the next section, we define two varieties of *fm*-algebra – S-algebra and W-algebra (for *strong* and *weak* *fm*-algebra, respectively) – which have somewhat different axioms for Δ and $*$.

This definition is motivated by the following ideas about programs and program modifications that we wish to formalize with *fm*-algebra: P represents the set of all programs; M represents the set of all program-modifications, where program modifications are formalized as functions from programs to programs that obey the axioms listed below (axioms S1–S7 in the case of *S*-algebra and axioms W1–W7 in the case of *W*-algebra). The process of performing a modification on some program *base* is formalized as the application of a modification function *m* to *base* (*i.e.*, *m* (*base*) denotes the program obtained by performing modification *m* on program *base*). Operation $\Delta(a, base)$ produces a modification function that takes *base* to *a* (*i.e.*, $\Delta(a, base)$ yields a modification that, when applied to *base*, produces *a*). Operation $m_1 * m_2$ combines two modification functions m_1 and m_2 to give a new modification function. (The models of *fm*-algebra given in Sections 5 and 6 are based on these ideas about programs and program modifications.)

We define the ternary integration operator $\_[\![\_]\!]\_ : P \times P \times P \longrightarrow P$ of a functional-modification algebra as follows:

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

Remark: More generally, we can let Δ and $*$ be partial functions - not every pair of modifications might be meaningfully combined into a single modification; similarly, it might be impossible to talk of $\Delta(a,base)$ for every pair of programs *a* and *base*. However, these problems can be handled by adding error elements *infeasibleProgram* and *infeasibleModification* to P and M, respectively, and extending Δ and $*$ to be total functions.

3. Axiomatisation of functional-modification algebras

The above definition formalises our intuitive explanation of program integration. In this section we look at properties that we might reasonably expect of Δ and $*$, given our interpretation of these operations. These properties are listed as axioms below. *I* denotes the identity function from P to P. (Thus, it may be interpreted as the empty or null modification.)

Definition. An *S-algebra* is a functional-modification algebra that satisfies the following axioms:

S1. $\Delta(a,a) = I$

S2. $\Delta(a,base)(base) = a$

S3-S6. <M, $*$> is a join-semi-lattice with *I* as the least element. This expands into the following four axioms:

S3. $*$ is commutative.

S4. $*$ is idempotent.

S5. I is the identity element with respect to $*$.

S6. $*$ is associative.

S7. $\Delta(a[\![base]\!]b,base) = \Delta(a,base) * \Delta(b,base)$

Axioms S1-S6 can be justified intuitively. Axiom S7 is a formalisation of our intuitive expectation of the integrated program (*e.g.*, the axiom can be read as: $a[\![base]\!]b$ is the element x such that $\Delta(x,base)$ is the combination of $\Delta(a,base)$ and $\Delta(b,base)$). Axiom S7 is discussed again later.

In Section 4 we derive some simple properties of the integration operator of an S-algebra. We first consider a weaker form of axioms S3-S7 that is sufficient to derive these various properties. This weaker set of axioms is motivated by the following observation: as the definition of _[[_]]_ indicates, we are not interested in "combining" *arbitrary* elements of M, but only modifications to the *same base* program. More formally, define

$$M_{base} \triangleq \{ \Delta(a,base) : a \in P \}.$$

We are interested in combining two modifications only if both are elements of some M_{base}. Hence, the axioms S3-S7 above may be weakened as follows.

Definition. A *W-algebra* is a functional-modification algebra that satisfies the following axioms:

W1. $\Delta(a,a) = I$

W2. $\Delta(a,base)(base) = a$

W3-W7. For every $base \in P$, $<M_{base}, * >$ is a join-semi-lattice with I as the least element. This expands into the following five axioms:

W3. $\Delta(a,base) * \Delta(b,base) = \Delta(b,base) * \Delta(a,base)$.

W4. $\Delta(a,base) * \Delta(a,base) = \Delta(a,base)$.

W5. $I * \Delta(a,base) = \Delta(a,base) = \Delta(a,base) * I$.

W6. $(\Delta(a,base) * \Delta(b,base)) * \Delta(c,base) = \Delta(a,base) * (\Delta(b,base) * \Delta(c,base))$.

W7. $\Delta(a,base) * \Delta(b,base) \in M_{base}$.

We now show that axiom S2 (= W2) implies that the axioms S7 and W7 are equivalent.

Proposition. If an *fm*-algebra satisfies axiom S2, then it satisfies axiom S7 iff it satisfies axiom W7, (*i.e.*, if S2 holds, then $\Delta(a[\![base]\!]b,base) = \Delta(a,base) * \Delta(b,base) \Leftrightarrow \Delta(a,base) * \Delta(b,base) \in M_{base}$).

Proof.

=> This is trivial since $\Delta(a[\![base]\!]b,base) \in M_{base}$ by definition.

<= We have $\Delta(a,base) * \Delta(b,base) = \Delta(c,base)$, for some c.

$(\Delta(a,base) * \Delta(b,base))(base) = \Delta(c,base)(base)$

$a[\![base]\!]b = c$

□

In particular, every W-algebra satisfies axiom S7, and every S-algebra satisfies axiom W7; thus, every S-algebra is also a W-algebra.

4. Properties of _[[_]]_

The _[[_]]_ operator of a W-algebra satisfies the following properties.

Proposition. $a[[base]]b = b[[base]]a$.

Proof.

$$\begin{aligned} a[[base]]b &= (\Delta(a,base) * \Delta(b,base))(base) \\ &= (\Delta(b,base) * \Delta(a,base))(base) && \text{(axiom W3)} \\ &= b[[base]]a \end{aligned}$$

□

Proposition. $a[[base]]a = a$.

Proof.

$$\begin{aligned} a[[base]]a &= (\Delta(a,base) * \Delta(a,base))(base) \\ &= (\Delta(a,base))(base) && \text{(axiom W4)} \\ &= a && \text{(axiom W2)} \end{aligned}$$

□

Proposition. $a[[base]]base = a$.

Proof.

$$\begin{aligned} a[[base]]base &= (\Delta(a,base) * \Delta(base,base))(base) \\ &= (\Delta(a,base) * I)(base) && \text{(axiom W1)} \\ &= (\Delta(a,base))(base) && \text{(axiom W5)} \\ &= a && \text{(axiom W2)} \end{aligned}$$

□

Definition. The *simultaneous integration* of elements $x_1, x_2, ..., x_n$ with respect to element *base* is the element $(x_1[[base]]x_2,...,x_n)$ defined by

$$(x_1[[base]]x_2,...,x_n) \triangleq (\Delta(x_1,base) * \Delta(x_2,base) * \cdots * \Delta(x_n,base))(base)$$

Note: The above definition makes sense as long as $*$ is associative, at least when restricted to M_{base}.

Proposition. $(a_1[[base]]a_2,...,a_n)$ is symmetric in $a_1,...,a_n$.

Proof. Follows directly from axioms W3 and W6.

□

Proposition. $(x[[base]]y)[[base]]z = (x[[base]]y,z)$.

Proof.

$$\begin{aligned} (x[[base]]y)[[base]]z &= (\Delta(x[[base]]y,base) * \Delta(z,base))(base) \\ &= (\Delta(x,base) * \Delta(y,base) * \Delta(z,base))(base) && \text{(axiom S7)} \\ &= (x[[base]]y,z). \end{aligned}$$

□

As an immediate consequence, we get the following associativity theorem.

Theorem. $(x[[base]]y)[[base]]z = x[[base]](y[[base]]z) = (x[[base]]z)[[base]]y = x[[base]]y,z$.

Note that _[[_]]_ is a ternary operator. By saying that _[[_]]_ is associative we mean that for any *base* the curried binary operator _[[*base*]]_ is associative. We intuitively expect program integration to be associative: associativity justifies integrating multiple variants of a base program by performing a succession of two-variant integrations (in any order).

5. From integration to *fm*-algebras

The previous sections outline one possible way of constructing integration operators satisfying some simple properties from binary operators Δ and $*$ that form an S-algebra or W-algebra. A natural question that arises is: how general is this approach? Is it reasonable to assume that integration operators may always be constructed in such a fashion? In this section we show the generality of this approach by showing that any integration operator satisfying certain simple properties can be constructed in such a fashion. Assume that $\_[\_]\_:(P \times P \times P) \longrightarrow P$ is a ternary operator on P satisfying the following properties (we expect most integration operators to satisfy these properties):

L1. $a\,[base\,]b = b\,[base\,]a$.

L2. $base\,[base\,]a = a$.

L3. $a\,[base\,]a = a$.

L4. $(a\,[base\,]b)[base\,]c = a\,[base\,](b\,[base\,]c)$.

Define $\Delta:(P \times P) \longrightarrow (P \longrightarrow P)$ by currying $\_[\_]\_$ as below:

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

In the following subsections we consider two different definitions of $*$, the operator for combining modifications. In each case the corresponding $\_[\![\_]\!]\_$ operator is shown to be the same as $\_[\_]\_$. We show that Δ and the first version of $*$ yield a W-algebra. We then show that Δ and the second version of $*$ yield an S-algebra, assuming that $\_[\_]\_$ satisfies one additional property (stated below as L5).

5.1. Definition 1

Define $*$ to be function composition. Let M be the range of Δ closed under function composition.

Theorem. $a[\![base]\!]b = a\,[base\,]b$

Proof.

$$\begin{aligned} a[\![base]\!]b &= (\Delta(a,base) * \Delta(b,base))(base) \\ &= \Delta(a,base)\,(\Delta(b,base)\,(base)) \\ &= a\,[base\,](b\,[base\,]base) \\ &= a\,[base\,]b \end{aligned}$$

□

Theorem. *<P,M,Δ,* > is a W-algebra.*

Proof.

W1. $$\begin{aligned} \Delta(a,a) &= \lambda b.\,(a\,[a\,]b) \\ &= \lambda b.b \\ &= I \end{aligned}$$

W2. $$\begin{aligned} \Delta(a,base)(base) &= a\,[base\,]base \\ &= a \end{aligned}$$

W3. $$\begin{aligned} \Delta(a,base) * \Delta(b,base) &= (\lambda c.\ a\,[base\,]c) * (\lambda c.\ b\,[base\,]c) \\ &= \lambda c.\ a\,[base\,](b\,[base\,]c) \\ &= \lambda c.\ (a\,[base\,]b)[base\,]c \\ &= \lambda c.\ (b\,[base\,]a)[base\,]c \\ &= \Delta(b,base) * \Delta(a,base) \end{aligned}$$

W4. $\Delta(a,base) * \Delta(a,base) = (\lambda b.\ a\,[base]\,b) * (\lambda b.\ a\,[base]\,b)$
$= \lambda b.\ a\,[base]\,(a\,[base]\,b)$
$= \lambda b.\ (a\,[base]\,a)[base]\,b$
$= \lambda b.\ a\,[base]\,b$
$= \Delta(a,base)$

W5. I is the identity with respect to function composition.

W6. Function composition is associative.

S7. $\Delta(a[\![base]\!]b,base) = \Delta(a\,[base]\,b,base)$
$= \lambda c.\ (a\,[base]\,b)[base]\,c$
$= \lambda c.\ a\,[base]\,(b\,[base]\,c)$
$= (\lambda c.\ a\,[base]\,c) * (\lambda c.\ b\,[base]\,c)$
$= \Delta(a,base) * \Delta(b,base)$

W7. This follows directly from S7 above.

□

5.2. Definition 2

We assume in this subsection that _[_]_ satisfies the following property, in addition to the properties L1-L4 listed at the beginning of this section:

L5. $(a\,[base]\,c)\ [c]\ (c\,[base]\,b) = (a\,[base]\,b)[base]\,c$.

Property L5 says that $(a\,[base]\,c)\ [c]\ (c\,[base]\,b)$ is another correct way of integrating three variants a, b, and c with respect to *base*. Define $*$ as follows.

$$m_1 * m_2 \triangleq \lambda p.\ m_1(p)\ [p]\ m_2(p).$$

Let M be the range of Δ closed under $*$.

Theorem. $a[\![base]\!]b = a\,[base]\,b$

Proof.

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

□

Theorem. $<P,M,\Delta,*>$ *is an S-algebra.*

Proof.

S1. $\Delta(a,a) = I$ (as in the previous subsection)

S2. $\Delta(a,base)(base) = a$ (as in the previous subsection)

S3. $m_1 * m_2 = \lambda p.\ m_1(p)\ [p]\ m_2(p)$
$= \lambda p.\ m_2(p)\ [p]\ m_1(p)$
$= m_2 * m_1$

S4. $m * m = \lambda p.\ m(p)\ [p]\ m(p)$
$= \lambda p.\ m(p)$
$= m$

S5. $m * I = \lambda p.\ m(p)\ [p]\ I(p)$

$= \lambda p.\ m(p)\ [p]\ p$

$= \lambda p.\ m(p)$

$= m$

S6. $(m_1 * m_2) * m_3 = \lambda p.\ (m_1(p)\ [p]\ m_2(p))\ [p]\ m_3(p)$

$= \lambda p.\ m_1(p)\ [p]\ (m_2(p)\ [p]\ m_3(p))$

$= m_1 * (m_2 * m_3)$

S7. $\Delta(a[\![base]\!]b, base) = \Delta(a\,[base\,]b, base)$

$= \lambda c.\ (a\,[base\,]b)[base\,]c$

$= \lambda c.\ (a\,[base\,]c)\ [c\,]\ (c\,[base\,]b)$ (using property L5)

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

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

□

Reps [Reps90] defines an integration operator _[_]_ and shows that it satisfies properties L1-L5 listed in this section. (The definition of _[_]_ from [Reps90] is briefly summarised at the beginning of the next section.) Hence, the construction above shows the consistency of the axioms of S-algebra and W-algebra.

6. An example

In this section, we look at an example of constructing an integration algorithm from an *fm*-algebra. We develop the integration algorithm first proposed by Horwitz, Prins and Reps [Horwitz89] (the HPR algorithm) and later expressed in an algebraic framework by Reps [Reps90]. The algorithm makes use of a program representation called a *program dependence graph*[Kuck81, Ferrante87]. 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 containing all vertices on which s has a transitive flow or control dependence. A dependence graph 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). The algebraic framework is based a partial order $\leq$ on single-point slices that denotes the relation "is-a-subslice-of". Thus, 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). Given a set A of single-point slices, the *downwards–closure* of A, $DC(A)$, is defined as

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

It can be seen that $DC(A \cup B) = DC(A) \cup DC(B)$. A set A is said to be *downwards–closed* if $DC(A) = A$. The operation of *upwards–closure* is similarly defined as

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

In this framework, a program is represented by the set of all single-point slices of the program. P, the domain of program representations, is taken to be the set of all downwards-closed sets of single-point slices. Let $\top$ denote the set of all single-point slices, and $\bot$ the empty set. Let $\cup$, $\cap$, and $-$ denote the set-theoretic union, intersection and difference operators. P is closed with respect to $\cup$ and $\cap$. P is not closed under $-$, but is closed under the pseudo-difference operator $\dot{-}$ defined as follows:

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

P is also closed under a similar (dual) operator $\dot{+}$ defined as follows, where $\overline{\ }$ denotes the set-theoretic complement with respect to $\top$.

$$X \dot{+} Y = \overline{UC(Y - X)}$$

Reps [Reps90] shows that $(P, \cup, \cap, \dot{-}, \dot{+}, \top)$ is a double Brouwerian algebra and that the HPR integra-

tion algorithm can be represented by the ternary operator _[_]_ defined by:

$$a\,[base\,]b \triangleq (a \div base) \cup (a \cap base \cap b) \cup (b \div base)$$

Here we see how the approach outlined in Section 2 can be used to arrive at the same definition.

The HPR algorithm considers the following two types of modifications: addition of slices and deletion of slices. Consider the modification of *base* to *a*. The set difference $(a–base)$ represents the slices that have been added to the *base* program. This suggests the possibility of representing this "addition of $(a–base)$" by the function $\lambda p.p \cup (a–base)$. Unfortunately, this function does not necessarily map downwards-closed sets to downwards-closed sets and hence may not be an element of (P $\longrightarrow$ P). Similarly, $(base–a)$ represents the set of slices that have been deleted and we would like to represent this deletion by the function $\lambda p.p–(base–a)$. This function too may not be an element of (P $\longrightarrow$ P). In what follows, we look at two different ways of changing these functions so that they do map downwards-closed sets to downwards-closed sets.

6.1. Definition 1

(Note that in what follows *a*, *base* and *p* are downwards-closed sets.) The addition of $(a–base)$ can be represented by the function $\lambda p.DC\,(p \cup (a–base)) = \lambda p.p \cup DC\,(a–b) = \lambda p.p \cup (a \div base)$. Thus, the addition of $(a–base)$ is interpreted as the addition of $DC\,(a–base)$. The deletion of $(base–a)$ can be represented by the function $\lambda p.DC\,(p–(base–a)) = \lambda p.DC\,(p \cap \overline{(base \cap \bar{a})}) = \lambda p.DC\,(p \cap (\overline{base} \cup a)) = \lambda p.DC\,((p \cap \overline{base}) \cup (p \cap a)) = \lambda p.DC\,(p–base) \cup (p \cap a) = \lambda p.(p \div base) \cup (p \cap a)$. The combination of the addition and deletion can be represented by the composition of the above two functions. (By applying the laws that relate $\cup$, $\cap$, and $\div$ in Brouwerian algebra, it can be shown that these two functions commute with each other with respect to $\circ$.) Thus, we arrive at the following definition of Δ:

$$\Delta\,(a,base) \triangleq (\lambda p.p \cup (a \div base)) \circ (\lambda p.(p \div base) \cup (p \cap a)).$$

We define $*$ to be function composition. Let M be the closure of the range of Δ under $\circ$. Then, it can be shown that $<$P,M,Δ,$*>$ is a W-algebra whose integration operator _[[_]]_ is the same as the integration operator in Brouwerian algebra. A short way of demonstrating this is to see that the above definition of Δ simplifies to

$$\begin{aligned}
\Delta\,(a,base) &= \lambda p.(a \div base) \cup (p \cap a) \cup (p \div base) \\
&= \lambda p.(a \div base) \cup (p \cap a \cap base) \cup ((p \cap a) \div base) \cup (p \div base) \\
&\quad \text{(since } x = (x \div y) \cup (x \cap y)\text{)} \\
&= \lambda p.(a \div base) \cup (p \cap base \cap a) \cup (p \div base) \\
&\quad \text{(since } (p \cap a) \div base \subseteq (p \div base)\text{)} \\
&= \lambda p.a\,[base\,]p
\end{aligned}$$

Hence, the desired result follows from the results of Section 5.

6.2. Definition 2

In Section 6.1, the addition of $(a–base)$ was interpreted as the addition of $DC\,(a–base)$, giving us the function $\lambda p.p \cup (a \div base)$. Analogously, the deletion of $(base–a)$ can be interpreted as the deletion of $UC\,(base–a)$, since the absence of $(base–a)$ in any program automatically implies the absence of $UC\,(base–a)$. Hence, this deletion can be represented by the function $\lambda p.p–UC\,(base–a) = \lambda p.p \cap \overline{UC\,(base–a)} = \lambda p.p \cap (a \div base)$. Any modification can be considered to be a combination of addi-

tions and deletions. This suggests the following definitions. The pair $\langle x,y\rangle$ will be used to denote the function $\lambda z.(z \cup x) \circ \lambda z.(z \cap y)$. Note that $\langle x,y\rangle = \langle x, y \cup x\rangle$. A unique representation for the function $\lambda z.(z \cup x) \circ \lambda z.(z \cap y)$ is obtained by restricting our attention to pairs of the form $\langle x,y\rangle$ where $x \subseteq y$. Thus, let M denote $\{ \langle x,y\rangle : x, y \in P, x \subseteq y \}$. (Any pair $\langle x,y\rangle$ where $x \not\subseteq y$ is assumed to be an abbreviation for $\langle x, x \cup y\rangle$.) Δ is defined as follows:

$$\Delta(a,base) \triangleq \langle a \div base, a \div base \rangle.$$

The operator $*$ is defined as follows:

$$\langle x_1,y_1\rangle * \langle x_2,y_2\rangle \triangleq \langle x_1 \cup x_2, y_1 \cap y_2\rangle.$$

The motivation for the above definition is as follows. Consider the modification $\langle x_1,y_1\rangle$. The addition component of this modification is the function $\langle x_1,\top\rangle$, while the deletion component is the function $\langle \bot,y_1\rangle$. The two additions $\langle x_1,\top\rangle$ and $\langle x_2,\top\rangle$ composed (in either order) yield $\langle x_1 \cup x_2,\top\rangle$. Similarly, the two deletions $\langle \bot,y_1\rangle$ and $\langle \bot,y_2\rangle$ composed yield $\langle \bot, y_1 \cap y_2\rangle$. It can be verified that whenever $\langle x_1,y_1\rangle$ and $\langle x_2,y_2\rangle$ commute with respect to $\circ$, $\langle x_1,y_1\rangle \circ \langle x_2,y_2\rangle$ is $\langle x_1 \cup x_2, y_1 \cap y_2\rangle$. If the two do not commute with respect to $\circ$, then the "addition" performed by one of them conflicts with the "deletion" performed by the other. (*i.e.*, some slices get added by one of them, while the same slices get deleted by the other.) The above definition of $*$ resolves the conflict in favour of the "addition." This follows from the interpretation of the pair $\langle x,y\rangle$: the deletion is done first, followed by the addition. (Note that $f \circ g$ denotes the function $\lambda x.f(g(x))$.)

The definition of $\_[\![\_]\!]\_$ gives us the following:

$$\begin{aligned}
a[\![base]\!]b &= (\Delta(a,base) * \Delta(b,base))(base) \\
&= (\langle a \div base, a \div base\rangle * \langle b \div base, b \div base\rangle)(base) \\
&= \langle (a \div base) \cup (b \div base), (a \div base) \cap (b \div base)\rangle(base) \\
&= (a \div base) \cup ((a \div base) \cap base \cap (b \div base)) \cup (b \div base) \\
&= (a \div base) \cup (a \cap base \cap b) \cup (b \div base) \qquad \text{(since } (x \div y) \cap y = x \cap y\text{)} \\
&= a[base]b
\end{aligned}$$

Though this definition of Δ and $*$ yields the same integration operator as the definitions in Section 6.1, the Δ and $*$ themselves do not satisfy the same set of axioms: although operators Δ and $*$ satisfy axioms S1-S6, they do not satisfy axiom S7. (This suggests that studying sets of axioms weaker than S1-S7 might be potentially useful.)

7. Separating consecutive edits

Program integration deals with the problem of reconciling "competing" modifications to a base program. A different, but related, problem is that of separating *consecutive* edits to a base program into individual edits on the base program. Consider the case of two consecutive edits to a program *base*; let *a* be obtained by modifying *base*, and *c* be obtained by modifying *a*. By "separating consecutive edits," we mean creating a program *b* that includes the second modification but not the first.

One approach to separating consecutive edits, suggested in [Reps90], is based on the assumption that the desired program *b* should satisfy the equation $a[\![base]\!]b = c$. Thus, the problem is related to solving an equation of the form $a[\![base]\!]x = c$ for x. In what follows, we look at solutions of this equation in W-algebra.

Lemma. x is a solution to $a[\![base]\!]x = c$ iff $x = m(base)$ where $m \in M_{base}$ is a solution to the equation $\Delta(a,base) * m = \Delta(c,base)$.

Proof. Note that $x = m(base)$ for some $m \in M_{base}$ iff $\Delta(x,base) = m$.

=> Let $a[\![base]\!]x = c$.
Then, $\Delta(a[\![base]\!]x,base) = \Delta(c,base)$.
Hence $\Delta(a,base) * \Delta(x,base) = \Delta(c,base)$, using axiom S7.

<= $\Delta(a,base) * \Delta(x,base) = \Delta(c,base)$.
$\Delta(a[\![base]\!]x,base) = \Delta(c,base)$, using axiom S7.
$a[\![base]\!]x = c$, applying both sides to *base* and using axiom W2.
□

The above lemma shows that solving the equation $a[\![base]\!]x = c$ can be reduced to solving an equation of the form $p * y = q$ in the join-semi-lattice M_{base} of modifications to *base*. The following result concerns equations of the latter form.

Theorem. Let <M, * > be a join-semi-lattice. Let $\sqsubseteq$ denote the corresponding partial order on M. Let $p,q \in$ M. Then,

1. The equation $p * x = q$ has a solution iff q itself is a solution iff $p \sqsubseteq q$.
2. If x_1 and x_2 are solutions of $p * x = q$ and $x_1 \sqsubseteq y \sqsubseteq x_2$, then y is a solution too.
3. If x_1 and x_2 are solutions of $p * x = q$, then $x_1 * x_2$ is a solution too.
4. The solutions to $p * x = q$ form a sub-semi-lattice of M with q as the maximum element (assuming that a solution exists).

Proof.

1. Let x satisfy $p * x = q$. Then,

$$p * q = p * (p * x) = (p * p) * x = p * x = q$$

Hence, q is a solution to $p * x = q$. By the definition of $\sqsubseteq$, this is equivalent to $p \sqsubseteq q$.

2. $x_1 \sqsubseteq y \sqsubseteq x_2$. Hence, $p * x_1 \sqsubseteq p * y \sqsubseteq p * x_2$. *i.e.*, $q \sqsubseteq p * y \sqsubseteq q$. Hence, y is a solution.

3. Since $p * x_1 = q$, $x_1 \sqsubseteq q$. Similarly, $x_2 \sqsubseteq q$. Hence, $x_1 \sqsubseteq (x_1 * x_2) \sqsubseteq q$. It follows from 1 and 2 that $x_1 * x_2$ is a solution.

4. Follows from 3.

Theorem.

1. $a[\![base]\!]x = c$ has a solution iff c itself is a solution.
2. If x_1 and x_2 are solutions to $a[\![base]\!]x = c$ then, $x_1[\![base]\!]x_2$ is also a solution.

Proof

1. This follows directly from the previous results.

2. This follows from the previous results as below. Since x_1 and x_2 are solutions of $a[\![base]\!]x = c$, it follows from the first lemma that $\Delta(x_1,base)$ and $\Delta(x_2,base)$ are solutions of the equation $\Delta(a,base) * m = \Delta(c,base)$. From the second lemma, it follows that $\Delta(x_1,base) * \Delta(x_2,base)$ is also a solution to the latter equation. Hence, $(\Delta(x_1,base) * \Delta(x_2,base))(base)$,*i.e.*, $x_1[\![base]\!]x_2$ is a solution to $a[\![base]\!]x = c$ (from the first lemma). □

From the construction in the Section 5, it can be seen that the above theorem holds for any _[_]_ satisfying the properties L1-L4 (listed at the beginning of Section 5).

8. Relation to previous work

This paper has presented new techniques for studying program-integration algorithms. We introduced a new formalism for expressing integration, based on the following ideas:

(1) Program modifications are formalized as functions from programs to programs that obey a certain set of axioms (S1–S7 in the case of S-algebra and W1–W7 in the case of W-algebra).

(2) The process of performing a modification on some program *base* is formalized as the application of a modification function m to *base*.

(3) There is an operation $\Delta(a, base)$ that produces a modification function that maps *base* to a.

(4) There is an operation $m_1 * m_2$ that combines two modification functions m_1 and m_2 to give a new modification function.

The discussions in Section 4 concerning the associativity of integration and in Section 7 concerning the problem of separating consecutive edits show that *fm*-algebra is a useful abstraction for studying the algebraic properties of program integration.

The work most closely related to the work presented in this paper is [Reps90], which also uses algebra to study program integration. There, the set of downwards-closed sets of single-point slices is shown to form a Brouwerian algebra; an integration operation is defined for Brouwerian algebra; and the integration operation for the Brouwerian algebra of downwards-closed sets of single-point slices is shown to correspond to the HPR integration algorithm.

This paper makes use of the results of [Reps90] in two ways: first, Section 5 establishes the consistency of the axioms of W-algebra and S-algebra by constructing models for them based on Brouwerian algebra and its integration operation; second, the constructions in Sections 5 and 6 show that the integration operation in a Brouwerian algebra—and hence, by the correspondence established in [Reps90], the HPR integration algorithm as well—corresponds to the integration operation in appropriately defined *fm*-algebras.

However, the S- and W-algebras studied in this paper are substantially different from Brouwerian algebra. Brouwerian algebra deals with one kind of element (which in [Reps90] formalizes the space of program representations). In contrast, *fm*-algebra has two kinds of elements; in our context, the two kinds of elements formalize the space of program representations and the space of program modifications. In other words, *fm*-algebra explicitly formalizes the notion of a *modification*, for which there is no corresponding notion in Brouwerian algebra. (A novelty of *fm*-algebra is that modifications are treated as functions.) Finally, Brouwerian algebra and *fm*-algebra deal with different sets of operators—$\sqcup$, $\sqcap$, and $\div$ in the case of Brouwerian algebra versus Δ and $*$ in the case of *fm*-algebra.

REFERENCES

Ferrante87.
Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* 9(3) pp. 319-349 (July 1987).

Horwitz89a.
Horwitz, S., Pfeiffer, P., and Reps, T., "Dependence analysis for pointer variables," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices,* (1989).

Horwitz89.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

Horwitz90.
Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).

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).

McKinsey46.
McKinsey, J.C.C. and Tarski, A., "On closed elements in closure algebras," *Annals of Mathematics* **47**(1) pp. 122-162 (January 1946).

Ottenstein84.
Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

Reps90.
Reps, T., "Algebraic properties of program integration.," 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).

Weiser84.
Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).

Yang90.
Yang, W., Horwitz, S., and Reps, T., "A program integration algorithm that accommodates semantics-preserving transformations," To appear in *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, (Irvine, CA, December 3-5, 1990), *ACM SIGSOFT Software Engineering Notes*, (1990).