ALGEBRAIC PROPERTIES OF PROGRAM INTEGRATION*

Thomas REPS
Computer Sciences Department, University of Wisconsin–Madison, 1210 W. Dayton Street, Madison, WI 53706

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 *behavior*, rather than the *text*, of a program variant is significant and must be incorporated 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. To determine this information, the algorithm employs a program representation that is similar to the *program dependence graphs* that have been used previously in vectorizing and parallelizing compilers.

This paper studies the algebraic properties of the program-integration operation, such as whether there are laws of associativity and distributivity. (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.") To answer such questions, we reformulate the Horwitz-Prins-Reps integration algorithm as an operation in a *Brouwerian algebra* constructed from sets of dependence graphs. (A Brouwerian algebra is a distributive lattice with an operation $a \dot{-} b$ characterized by $a \dot{-} b \sqsubseteq c$ iff $a \sqsubseteq b \sqcup c$.) In this algebra, the program-integration operation can be defined solely in terms of $\sqcup$, $\sqcap$, and $\dot{-}$. By making use of the rich set of algebraic laws that hold in Brouwerian algebras, we have established a number of the integration operation's algebraic properties.

1. Introduction

1.1. The Program-Integration Problem

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. Given a program P and a set of variants of P—created, say, by modifying separate copies of P—the goal is to determine whether the modifications interfere, and, if they do not, to create an *integrated* program that includes all changes as well as all features of P that are preserved in all variants [10]. Opportunities for program integration arise in many situations:

(1) A system may be "customized" by a user while simultaneously being upgraded by a maintainer. When the next release of the system is sent to the user, he must integrate his customized version of the system and the newly released version with respect to the earlier release so as to incorporate both his customizations and the upgrades.

(2) While systems are being created, program development is often a cooperative activity that involves multiple programmers. If a task can be decomposed into independent pieces, the different aspects of the task can be developed and tested independently by different programmers. However, if such a

*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.

To appear in *Science of Computer Programming*.

A preliminary version of this work appeared in *Proceedings of the 3rd European Symposium on Programming* (Copenhagen, Denmark, May 15-18, 1990), *Lecture Notes in Computer Science*, Vol. 432, N. Jones (ed.), Springer-Verlag, New York, NY, 1990 [26].

decomposition is not possible, the members of the programming team must work with multiple, separate copies of the source files, and the different versions of the files must ultimately be integrated to produce a common version.

(3) Suppose a tree or dag of related versions of a program exists (to support different machines or different operating systems, for instance), and the goal is to make the same enhancement or bug-fix to all of them. For example, if the change is made to the root version—by manually modifying a copy of the root program—the process of installing the change in all other versions requires a succession of program integrations.

Anyone who has had to reconcile divergent lines of development will recognize these situations and appreciate the need for automatic assistance.

At present, the only available tools for integration implement an operation for merging files as strings of text, such as the UNIX[1] utility *diff3*. This approach has the advantage that the current tools are as applicable to merging documents, data files, and other text objects as they are to merging programs. However, these tools are necessarily of limited utility for integrating programs because the manner in which two programs are merged is not *safe*—one has no guarantees about the way the program that results from a purely *textual* merge behaves in relation to the behaviors of the programs that are the arguments to the merge. For example, if one variant contains changes only on lines 5–10, while the other variant contains changes only on lines 15–20, *diff3* would deem these changes to be interference-free; however, just because changes are made at different places in a program is no reason to believe that the changes are free of undesirable interactions. The merged program produced by such a tool must, therefore, be checked carefully for conflicts that might have been introduced by the merge.[2]

Our goal is to design a *semantics-based* tool for program integration. We want a tool that—given program *Base* and two variants *A* and *B*—makes use of knowledge of the programming language to determine whether the changes made to *Base* to produce *A* and *B* have undesirable semantic interactions; only if there is no such interference should the tool produce a merged program *M*.

While our long-term goal is to design such a tool for a full-fledged programming language, for now we are using a simplified model of the program-integration problem so as to make it amenable to theoretical study. This model possesses the essential features of the problem, and thus permits us to conduct our studies without being overwhelmed by inessential details. Our integration model has the following characteristics:

(1) We restrict our attention to the integration of programs written in a simple programming language that has only assignment statements, conditional statements, while loops, and final output statements (called *end* statements); by definition, only those variables listed in the end statement have values in the final state. The language does not include input statements; however, a program can use a variable before assigning to it, in which case the variable's value comes from the initial state.

(2) When an integration algorithm is applied to base program *Base* and variant programs *A* and *B*, and if integration succeeds—producing program *M*—then for any initial state σ on which *Base*, *A*, and *B*

---

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

[2]Several managers in industry have told me that their mechanism to avoid integration conflicts is based on the modular structure of systems. They assign overall responsibility for a given module of a system to a particular programmer, and institute a policy that any changes to a module must be cleared with the person responsible. However, the notion that module boundaries protect against interference—even in conjunction with the above policy—is as flawed as the notion used in *diff3*. Both rely the incorrect assumption that "disjoint changes are interference free."

all terminate normally,[3] *M* must have the following properties:

(i) *M* terminates normally on σ.

(ii) For any variable *x* that has final value *v* after executing *A* on σ, and either no final value or a different final value *v′* after executing *Base* on σ, *x* has final value *v* after executing *M* on σ (*i.e.*, *M* agrees with *A* on *x*).

(iii) For any variable *y* that has final value *v* after executing *B* on σ, and either no final value or a different final value *v′* after executing *Base* on σ, *y* has final value *v* after executing *M* on σ (*i.e.*, *M* agrees with *B* on *y*).

(iv) For any variable *z* that has the same final value *v* after executing *Base*, *A*, and *B* on σ, *z* has final value *v* after executing *M* on σ (*i.e.*, *M* agrees with *Base*, *A*, and *B* on *z*).

(3) Program *M* is to be created only from components that occur in programs *Base*, *A*, and *B*.

A more informal statement of Property (2) is: changes in the behavior of *A* and *B* with respect to *Base* must be incorporated in the integrated program, along with the unchanged behavior of all three.

Properties (1) and (3) are syntactic restrictions that limit the scope of the integration problem. Property (2) defines the model's *semantic* criterion for integration and interference. *Any* program *M* that satisfies Properties (1), (2), and (3) integrates *Base*, *A*, and *B*; if no such program exists then *A* and *B* interfere with respect to *Base*. However, Property (2) is not decidable, even under the restrictions given by Properties (1) and (3); consequently, any program-integration algorithm will sometimes fail to produce an integrated program—and report interference—even though there is actually *no* interference (*i.e.*, even when there is *some* program that meets the criteria given above).

1.2. The Horwitz-Prins-Reps Algorithm for Program Integration

The first algorithm that meets the requirements given above was given by Horwitz, Prins, and Reps in [10]. Thus, that algorithm—referred to hereafter as the HPR algorithm—is the first algorithm for semantics-based program-integration.

The HPR algorithm represents a fundamental advance over text-based program-integration algorithms, and provides the first step in the creation of a theoretical foundation for building a semantics-based program-integration tool. Changes in *behavior* (in the sense of Property (2) above) rather than changes in text are detected, and are incorporated in the integrated program. Although it is undecidable to determine whether a program modification actually leads to a change in program behavior, it is possible to determine a safe approximation by comparing each of the variants with the original program *Base*. To determine this information, 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 [5, 14]. It makes use of an operation on these graphs called *program slicing* [19, 28] to find potentially changed computations. The HPR algorithm is summarized in Section 2, which also presents an example of an integration. (Full details—and a more extensive example—can be found in [10].)

This paper concerns not the HPR algorithm, but a close relative of it. (The revised integration algorithm is presented in Section 3.) We investigate the algorithm's algebraic properties, which are of particular interest when dealing with compositions of integrations. For example, if three variants of a given base program are to be integrated by a pair of (two-variant) integrations, it is important to know whether there is a law of associativity to guarantee that it does not matter which two variants are integrated first. (Such a law does hold; it is formulated as Theorem 4.4 and proven in Section 4.2.)

---

[3] There are two ways in which a program may fail to terminate normally on some initial state: (1) the program contains a non-terminating loop, or (2) a fault occurs, such as division by zero.

Although the capabilities of our current integration algorithms are severely limited, recent research has made progress towards extending the set of language constructs to which they apply [9, 11]. Our hope is that such extensions will share with the basic integration algorithm a common set of algebraic properties. However, we would like to avoid having to re-prove that each property holds every time we enhance our techniques. Instead, we would like to have a framework that would not only let us establish the algebraic properties of program integration, but would also allow us to show that a new algorithm possesses these properties merely by demonstrating that the algorithm meets the conditions of the framework. This paper uses lattice theory to provide such a framework.

1.3. An Overview of the Contents

A novel feature of our study is the use of *Brouwerian algebra*, rather than, for example, Boolean algebra or relational algebra. A Brouwerian algebra [16] is a distributive lattice with a *pseudo-difference* operation, $a \dot{-} b$, characterized by $a \dot{-} b \sqsubseteq c$ iff $a \sqsubseteq b \sqcup c$ (see Section 3). The connection between program integration and Brouwerian algebra is made as follows: we introduce a Brouwerian algebra constructed from sets of dependence graphs; in this algebra, the program-integration operation can be expressed solely in terms of the operations $\sqcup$, $\sqcap$, and $\dot{-}$.

The contributions of this paper can be summarized as follows:

(1) It establishes a number of *algebraic properties* that hold for the integration operation. These investigations make use of the rich set of algebraic laws that hold in Brouwerian algebras.

(2) It provides a *lattice-theoretic framework* for studying the common properties of different integration algorithms. The operation we define to integrate elements of a Brouwerian algebra is expressed purely in terms of $\sqcup$, $\sqcap$, and $\dot{-}$, and thus has an analogue in all Brouwerian algebras. The properties of the integration operation are established using only algebraic identities and inequalities, and thus the results obtained hold for all Brouwerian algebras. Consequently, to show that a proposed program-integration algorithm shares these properties, one merely has to show that the algorithm can be formulated as an integration operation in some Brouwerian algebra.

(3) It identifies a *new criterion* for program integration, based on the operation in our framework that is the *dual* of the integration operation.

The paper is divided into seven sections and three appendices. Section 2 provides an overview of the HPR algorithm for program integration. It also reviews the results that were established in [22, 25] concerning the semantic properties of a program that results from an integration. Readers familiar with the HPR algorithm can skip directly to Section 3, which introduces the concepts from lattice theory on which this paper's results are based. Section 3.1 discusses the considerations that lead us to reformulate the HPR algorithm as an operation in a Brouwerian algebra. Section 3.2 defines a lattice constructed from sets of dependence-graph slices and shows that it forms a Brouwerian algebra. Section 3.3 discusses the relationship between Brouwerian and Boolean algebras. Section 3.4 defines the operation to integrate elements of a Brouwerian algebra. Section 3.5 discusses how this operation—in the lattice of dependence-graph slice sets defined in Section 3.2—relates to the HPR algorithm.

In Section 4, the algebraic framework from Section 3 is used to pose and settle three questions concerning properties of the integration operation. Section 4.2 gives the proof of an associative law for the integration operation; it also defines a generalization of the integration operation to one that simultaneously integrates more than two variants with a given base element. Section 4.3 addresses the question of whether there is an integrand compatible with a given base *base*, integrand *a*, and result *m*. This problem is related to one of the applications of program integration, that of separating consecutive edits on some base program into individual edits on the base program. Section 4.4 concerns a similar question; it looks

at the question of whether there is a base element that is compatible with given integrands *a* and *b*, and result *m*.

Section 5 concerns *double* Brouwerian algebras—Brouwerian algebras whose duals are also Brouwerian algebras. It introduces the *quotient* operation, which is the dual of the pseudo-difference operation. In Section 5.2, it is shown that the lattice of dependence-graph slice sets from Section 3.2 is a double Brouwerian algebra. Section 5.3 concerns the operation that is the dual of the integration operation, and shows how the two operations are related. Section 5.4 re-examines the question of when there is an integrand compatible with a given base *base*, integrand *a*, and result *m* and extends the result from Section 4.3.

Section 6 concerns more pragmatic issues. Section 6.1 describes a system that implements the ideas discussed in the paper. Section 6.2 describes how the ideas from this paper may make it possible to eliminate a restriction that is part of the HPR algorithm. The HPR algorithm assumes that a special program editor is used to create the program variants from the base program: the editor provides a tagging capability so that common statements and predicates can be identified in different versions. (The assumption is stated fully at the beginning of Section 2.2.) As discussed in Section 6.2, it is possible to construct Brouwerian algebras whose elements are sets of dependence graphs that do *not* have tags on their components. Using sets of untagged dependence graphs would eliminate the necessity of supporting program integration by a closed system; instead, it would be possible to handle programs created using ordinary text editors. (The drawback of this approach is that it entails additional costs for finding the program that corresponds to the set of dependence graphs that result from an integration.)

Section 7 discusses the relationship of the work described in this paper to previous work on program integration.

To make the paper self-contained, there are three appendices that concern the basic algebraic laws that hold for elements of Brouwerian algebras. Appendix A covers the algebraic laws that hold among the operations $\sqcup$, $\sqcap$, and $\dot{-}$ in a Brouwerian algebra. Appendix B concerns the basic algebraic laws for the operation to integrate elements of a Brouwerian algebra. Appendix C gives a few laws that relate the operations $\sqcup$, $\sqcap$, $\dot{-}$, and $\div$ in a double Brouwerian algebra. Thus, it may be helpful to scan the appendices in conjunction with Sections 3, 4, and 5. (Throughout the body of the paper, when a law given in one of the appendices is used to justify a step of a proof, it is referred to by the number given for the law in the appropriate appendix.)

2. Overview of the HPR Algorithm for Program Integration

This section provides an overview of the HPR algorithm, which uses program dependence graphs to integrate programs [10]. It summarizes parts of [10], which contains a comprehensive description of the HPR algorithm. This summary is presented here because it is the starting point from which our new techniques are developed.

The integration of *A* and *B* with respect to *Base* requires combining three structures: $\Delta(A, Base)$, $\Delta(B, Base)$, and $Pre(A, Base, B)$, where $\Delta(A, Base)$ and $\Delta(B, Base)$ represent potentially changed computations of *A* and *B* with respect to *Base*, respectively, and $Pre(A, Base, B)$ represents computations that are preserved in all three. To determine this information, the HPR algorithm employs graphs that represent the dependences between program elements.

2.1. Program Dependence Graphs and Program Slicing

The program dependence graphs used by the HPR algorithm are similar to those used previously for representing programs in vectorizing and parallelizing compilers [5, 14]. Different definitions of program dependence representations have been given, depending on the intended application, but all are variations

on a theme introduced in [13] and share the common feature of having an explicit representation of data dependences (see below). The "program dependence graphs" defined in [5] introduced the additional feature of an explicit representation for control dependences (see below).[4]

Definition 2.1. A *directed graph* G consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$. Each edge $b \longrightarrow c \in E(G)$, where $b, c \in V(G)$, is directed from b to c; we say that b is the *source* of the edge and that c is the *target*.

As explained below, the vertices and edges of the directed graphs used in this paper are also labeled with some additional information.

Definition 2.2. A *program dependence graph* (or PDG) for program P is a directed graph whose vertices are connected by several kinds of edges. The assignment statements and control predicates of P are represented by vertices of the graph; these vertices are labeled with the text of the associated statement or predicate. In addition, there are three other kinds of vertices (which are labeled appropriately): there is a distinguished vertex called the *entry vertex*; there is an *initial-definition vertex* for each variable that may be used before being defined; and there is a *final-use vertex* for each variable named in P's end statement. There are four kinds of edges in a PDG: *control dependence edges*, *loop-independent flow dependence edges*, *loop-carried flow dependence edges*, and *def-order dependence edges*. (See Definitions 2.3, 2.4, and 2.5.) The latter three kinds of edges are referred to collectively as the graph's *data dependence edges*.

Example. Figure 1 shows an example program and its program dependence graph.

The source of a control dependence edge is either the entry vertex or a predicate vertex and each edge is labeled either **true** or **false**. A control dependence edge from vertex v to vertex w means (roughly) that during execution, whenever the predicate represented by v is evaluated and its value matches the label on the edge to w, then the program component represented by w will eventually be executed.[5]

Definition 2.3. A program dependence graph for program P contains a *control dependence edge* with source v and target w, denoted by $v \longrightarrow_c w$, iff one of the following holds:

(1) v is the entry vertex, and w represents a component of P that is not nested within any control predicate; these edges are labeled **true**.

(2) v represents a control predicate, and w represents a component of P nested immediately within the control construct whose predicate is represented by v. If v is the predicate of a while-loop, the edge $v \longrightarrow_c w$ is labeled **true**; if v is the predicate of a conditional statement, the edge $v \longrightarrow_c w$ is labeled **true** or **false** according to whether w occurs in the **then** branch or the **else** branch, respectively.

A data dependence edge with source v and target w means (roughly) that the program's behavior might change if the relative order of the components represented by v and w were reversed.

---

[4]The definition of program dependence graph given here (which is taken from [10]) differs from that of [5] in two ways. First, our definition covers only the restricted language described earlier, and hence is less general than the one given in [5]. Second, because of the particular needs of the program-integration problem, we omit certain classes of data dependence edges and define one additional class. However, the structures we define and those defined in [5] share the feature of explicitly representing both control and data dependences; for this reason, despite their differences, we refer to our graphs as "program dependence graphs," borrowing the term from [5].

[5]A method for determining control dependence edges for arbitrary programs is given in [5]; however, because we are assuming that programs include only assignment, conditional, and while statements, the control dependence edges can be determined in a much simpler fashion. For the language under consideration here, the control dependence edges essentially represent the program's nesting structure. This simplification is reflected in Definition 2.3.

```
program
   sum := 0;
   x := 1;
   while x < 11 do
      sum := sum + x;
      x := x + 1
   end
end(x, sum)
```

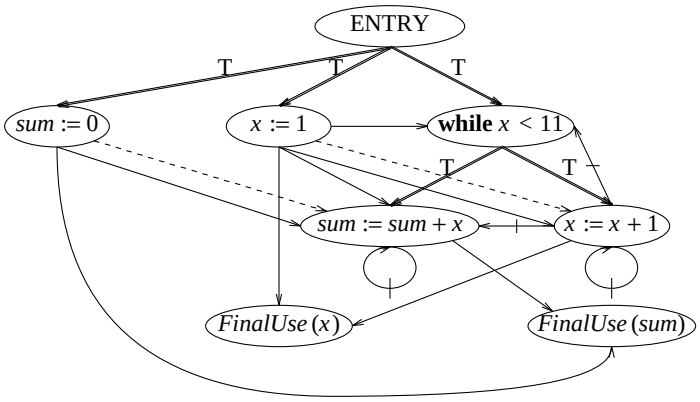


Figure 1. An example program, which sums the integers from 1 to 10 and leaves the result in the variable *sum*, and its program dependence graph. The boldface arrows represent control dependence edges, solid arrows represent loop-independent flow dependence edges, solid arrows with a hash mark represent loop-carried flow dependence edges, and dashed arrows represent def-order dependence edges.

Definition 2.4. A program dependence graph for program P contains a *flow dependence edge* with source v and target w, denoted by $v \longrightarrow_f w$, iff v represents an assignment to a variable x and w represents a use of x reached by v. The edge is *loop carried*, denoted by $v \longrightarrow_{lc(p)} w$, if v and w are both nested within a loop L, the predicate vertex of loop L is p, and v reaches w along a path in the control-flow graph for program P that includes the back-edge of loop L; otherwise the edge is *loop independent*, denoted by $v \longrightarrow_{li} w$. (Initial definitions of variables are considered to occur at the beginning of the control-flow graph for P, and final uses of variables are considered to occur at the end of the graph.)

Definition 2.5. A program dependence graph for program P contains a *def-order dependence edge* with source v, target w, and *witness* u, denoted by $v \longrightarrow_{do(u)} w$, iff (1) v and w both represent assignments to the same variable x, (2) there exist flow dependences $v \longrightarrow_f u$ and $w \longrightarrow_f u$, (3) v and w are in the same branch of any conditional statement that encloses both of them, and (4) v lexically precedes w.

Definition 2.6. Let s be a vertex of 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 (*i.e.*, all vertices that can reach s via flow or control edges):

$$V(G/s) \triangleq \{ w \in V(G) \mid w \longrightarrow^*_{c,f} s \}.$$

We extend the definition to a set of vertices $S = \bigcup_i s_i$ as follows:

$$V(G/S) = V(G/(\bigcup_i s_i)) \triangleq \bigcup_i V(G/s_i).$$

It is useful to define $V(G/v) = \emptyset$ for any $v \notin V(G)$. The edges in the graph G/S are essentially those in the subgraph of G induced by $V(G/S)$, with the exception that a def-order edge $v \longrightarrow_{do(u)} w$ is only included if, in addition to v and w, $V(G/S)$ also contains the witness-vertex u. In terms of the four types of edges in a program dependence graph we have:

$$
\begin{aligned}
E(G/S) \triangleq \quad & \{ (v \longrightarrow_c w) \in E(G) \mid v, w \in V(G/S) \} \\
& \cup \{ (v \longrightarrow_{li} w) \in E(G) \mid v, w \in V(G/S) \} \\
& \cup \{ (v \longrightarrow_{lc} w) \in E(G) \mid v, w \in V(G/S) \} \\
& \cup \{ (v \longrightarrow_{do(u)} w) \in E(G) \mid u, v, w \in V(G/S) \}.
\end{aligned}
$$

Example. Figure 2 shows the graph that results from slicing the program dependence graph from Figure 1 with respect to the final-use vertex for *x*, together with the program to which it corresponds.

The significance of a slice is that it captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice [22]. In our case a program point can be (1) an assignment statement, (2) a control predicate, or (3) a final use of a variable in an end statement. Because a statement or control predicate can be reached repeatedly in a program, by "computing the same sequence of values for each element of the slice" we mean the following: (1) for any assignment statement the same *sequence* of values are assigned to the target variable; (2) for a predicate the same *sequence* of boolean values are produced; and (3) for each final use the same value for the variable is produced.

Theorem 2.7. (Slicing Theorem [22]). *Let Q be a slice of program P with respect to a set of vertices. If* σ *is a state on which P halts, then for any state* σ' *that agrees with* σ *on all variables for which there are initial-definition vertices in* G_Q*: (1) Q halts on* σ'*, (2) P and Q compute the same sequence of values at each program point of Q, and (3) the final states agree on all variables for which there are final-use vertices in* G_Q.

2.2. An Algorithm for Integrating Programs

One of the requirements of the HPR algorithm is that program components (*i.e.*, statements and predicates) must be tagged so that corresponding components can be identified in all three versions. Component tags can be provided by a special editor that obeys the following conventions:

(1) When a copy of a program is made—*e.g.*, when a copy of *Base* is made in order to create a new variant—each component in the copy is given the same tag as the corresponding component in the

---

```
program
  x := 1;
  while x < 11 do
    x := x + 1
  end
end(x)
```

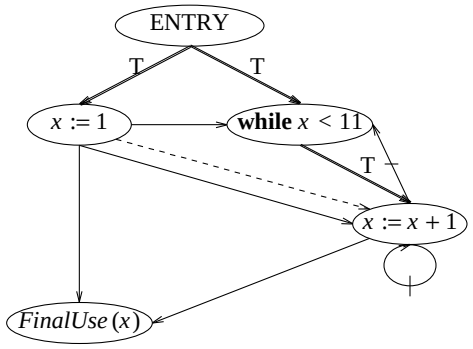


Figure 2. The graph that results from slicing the example from Figure 1 with respect to the final-use vertex for *x*, together with the program to which it corresponds.

original program.

(2) The operations on program components supported by the editor are insert, delete, and move. A newly inserted component is given a previously unused tag; the tag of a component that is deleted is never re-used; a component that is moved from one position to another retains its tag.

(3) The tags on components persist across different editing sessions and machines.

(4) Tags are allocated by a single server, so that two different editors cannot allocate the same new tag.

A tagging facility meeting these requirements can be supported by language-based editors, such as those that can be created by such systems as MENTOR [3], GANDALF [18], and the Synthesizer Generator[6]

Component tags furnish the means for identifying how the program-dependence-graph vertices in different versions correspond. It is the tags that are used to determine "identical" vertices when operations are performed using vertices from different program dependence graphs. For instance, when we speak below of "identical slices," where the slices are actually taken in different graphs (*e.g.*, $(G_{Base}/v) = (G_A/v)$), we mean that the slices are isomorphic under the mapping provided by the editor-supplied tags.

Remark. Except where we wish to emphasize which program components have the same tag, we do not indicate program-component tags in our examples. When we do indicate program-component tags, the tag is placed between square brackets to the left of the component (*e.g.*, [3] $z := y$). For all examples in which tags are not indicated explicitly, our convention is that components in different programs that have the same text also have the same tag.

The first step of the program-integration algorithm determines the slices of *A* and *B* that are changed from *Base* and the slices of *Base* that are preserved in both *A* and *B*; the second step combines these slices to form the merged graph G_M; the third step tests G_M for interference.

Step 1: Determining changed and preserved slices

Let G_{Base}, G_A, and G_B denote the program dependence graphs for programs *Base*, *A*, and *B*, respectively. By Theorem 2.7, if the slice of variant G_A at vertex v differs from the slice of G_{Base} at v, then G_A and G_{Base} might compute different values at v. In other words, vertex v is a site that potentially exhibits changed behavior in the two programs. Thus, we define the *affected points* of G_A with respect to G_{Base}, denoted by $AP_{A,\,Base}$, to be the subset of vertices of G_A whose slices in G_{Base} and G_A differ:

$$AP_{A,\,Base} \triangleq \{\, v \in V(G_A) \mid (G_{Base}/v) \neq (G_A/v) \,\}.$$

We define $AP_{B,\,Base}$ similarly. It follows that the slices $G_A/AP_{A,\,Base}$ and $G_B/AP_{B,\,Base}$ capture all the slices of *A* and *B* (respectively) that differ from *Base*, and so we make the following definitions:

$$\Delta(A, Base) \triangleq G_A/AP_{A,\,Base}$$
$$\Delta(B, Base) \triangleq G_B/AP_{B,\,Base}.$$

A vertex that has the same slice in all three programs is guaranteed to exhibit the same behavior. Thus, we define the *preserved points* of G_{Base}, denoted by $PP_{A,\,Base,\,B}$, to be the subset of vertices of G_{Base} with identical slices in G_{Base}, G_A, and G_B:

$$PP_{A,\,Base,\,B} \triangleq \{\, v \in V(G_{Base}) \mid (G_A/v) = (G_{Base}/v) = (G_B/v) \,\}.$$

The slices common to *Base*, *A*, and *B* (*i.e.*, the slices unchanged in both *A* and *B*) are captured by the slice $G_{Base}/PP_{A,\,Base,\,B}$, and so we make the following definition:

---

[6]The Synthesizer Generator is a Trademark of GrammaTech, Inc.

$$Pre(A, Base, B) \triangleq G_{Base} / PP_{A, Base, B}.$$

Stated another way, $Pre(A, Base, B)$ consists of the union of the slices that are identical in all three graphs.

Step 2: Forming the merged graph

Definition 2.8. Given directed graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$, whose vertices might have some tags in common, the *graph union* of G_A and G_B, denoted by $G_A \cup_g G_B$, is defined as

$$G_A \cup_g G_B \triangleq (V_A \cup V_B, E_A \cup E_B).$$

The merged graph G_M is formed by taking the graph union of the slices that characterize the changed behavior of *A*, the changed behavior of *B*, and behavior of *Base* preserved in both *A* and *B*:

$$G_M \triangleq \Delta(A, Base) \cup_g Pre(A, Base, B) \cup_g \Delta(B, Base).$$

Step 3: Testing for interference

There are two possible ways by which the graph G_M can fail to represent a satisfactory integrated program; we refer to them as "Type I interference" and "Type II interference." The criterion for Type I interference is based on a comparison of slices of G_A, G_B, and G_M. The slices $G_A / AP_{A, Base}$ and $G_B / AP_{B, Base}$ represent the changed slices of *A* and *B*, respectively. There is Type I interference if G_M does not preserve these slices; that is, there is Type I interference if either

$$(G_M / AP_{A, Base}) \neq (G_A / AP_{A, Base})$$

or

$$(G_M / AP_{B, Base}) \neq (G_B / AP_{B, Base}).$$

The final step of the HPR algorithm involves reconstituting a program from the merged program dependence graph. However, it is possible that there is no such program—the merged graph can be an infeasible program dependence graph; this is Type II interference. (The reader is referred to [10] for a discussion of reconstructing a program from the merged program dependence graph and the inherent difficulties of this problem.)

If neither kind of interference occurs, a program whose program dependence graph is G_M is returned as the result of the integration operation.

Example. The following example illustrates the HPR algorithm. The tags on statements are noted between square brackets.

A	*Base*	*B*	$\Delta(A, Base)$	$Pre(A, Base, B)$	$\Delta(B, Base)$	A [*Base*] B
program	**program**	**program**	$\emptyset$	**program**	**program**	**program**
[1] $x := 0$	[1] $x := 0$;	[1] $x := 0$;		[1] $x := 0$	[1] $x := 0$;	[1] $x := 0$;
end(x)	[2] $y := x$	[2] $y := x$;		**end**(x)	[2] $y := x$;	[2] $y := x$;
	end(x, y)	[3] $z := y$			[3] $z := y$	[3] $z := y$
		end(x, y, z)			**end**(z)	**end**(x, z)

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

2.3. Semantic Properties of the Integrated Program

The following theorem, Theorem 2.9, characterizes the execution behavior of the integrated program produced by the HPR algorithm in terms of the behaviors of the base program and the two variants [22, 25]. It shows that the integrated program produced by the HPR algorithm incorporates the changed behaviors of both variants *A* and *B* (with respect to base program *Base*) as well as the unchanged behavior of *A*, *B*, and *Base*. Thus, the HPR algorithm meets the semantic criterion of the integration model that was introduced in Section 1.1.

Theorem 2.9. (Integration Theorem [22, 25]). *If programs A and B are two non-interfering variants of Base, and program M is the result of integrating A and B with respect to Base, then for any initial state* σ *on which A, B, and Base all halt,*

(1) *M halts on* σ.

(2) If *x is a variable defined in the final state of A for which the final states of A and Base disagree, then the final state of M agrees with the final state of A on x.*

(3) If *y is a variable defined in the final state of B for which the final states of B and Base disagree, then the final state of M agrees with the final state of B on y.*

(4) If *z is a variable on which the final states of A, B, and Base agree, then the final state of M agrees with the final state of Base on z.*

3. Using Lattice Theory to Describe Program Integration

In unpublished work, Susan Horwitz and I found proofs of several algebraic properties of the HPR algorithm. However, because of the many different types of elements that occur in dependence graphs, the proofs by which these results were established were very involved, containing many sub-cases and argument by *reductio ad absurdum*.

This section introduces the means by which these complications are side-stepped. In motivating the new approach, it is necessary to introduce three different—but related—partially ordered sets. A close relative of the HPR integration algorithm is expressed as an operation in the last of these, which is a *Brouwerian algebra* constructed from sets of dependence-graph slices. This formulation allows us to give proofs of the integration algorithm's algebraic properties by simple manipulations of formulae; in these proofs, we make use of the rich set of algebraic laws—both identities and inequalities—that hold in Brouwerian algebras.

3.1. Motivation

To understand the considerations that lead us to reformulate the HPR algorithm as an operation in a Brouwerian algebra, consider how *Pre*(*A*, *Base*, *B*) was characterized in Section 2.2: "*Pre*(*A*, *Base*, *B*) consists of the slices that are identical in all three graphs." This terminology suggests that *Pre*(*A*, *Base*, *B*) is the meet of *A*, *Base*, and *B* in a lattice of dependence graphs ordered by "is-a-slice-of," the meet operation being "greatest common slice."[7]

---

[7] A *lattice* is an algebra $(L, \sqcup, \sqcap)$, where L is a set of elements that is closed under $\sqcup$ (join) and $\sqcap$ (meet), and for all a, b, and c in L the following axioms are satisfied:

$a \sqcup a = a$	$a \sqcap a = a$	$a \sqcup (a \sqcap b) = a$
$a \sqcup b = b \sqcup a$	$a \sqcap b = b \sqcap a$	$a \sqcap (a \sqcup b) = a$
$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$	$(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$	

The symbol $\sqsubseteq$ will be used to denote the partial order on the elements of L given by $a \sqsubseteq b$ iff $a \sqcap b = a$ (or, equivalently, $a \sqsubseteq b$ iff $a \sqcup b = b$). All lattices considered in this work have a least element and a greatest element, denoted by $\bot$ and $\top$, respectively, which satisfy the following axioms:

Definition 3.1. The symbol G will be used to denote the set of well-formed PDGs. We extend the definition of slicing with respect to a vertex set (see Definition 2.6) to that of *slicing with respect to a PDG* as follows: let a and b be two PDGs in G; the *slice* of a with respect to b, denoted by a/b, is

$$a/b \triangleq a/V(b).$$

We say that b is a *slice* of a iff $a/b = b$. The symbol $\leq$ will be used to denote the partial order "is-a-slice-of" on the elements of G (*i.e.*, $b \leq a$ iff b is a slice of a).

Observation. Let $a \wedge b$ denote the greatest common slice of dependence graphs a and b. The algebra (G, $\wedge$) is a meet semi-lattice.

Example. A portion of (G, $\wedge$) is illustrated in Figure 3. For instance, the greatest common slice of the elements labeled A and B in Figure 3 is the element labeled C. (Note that the variable list in C's end statement is empty.)

Remark. In Figure 3, as well as in other similar figures of the paper, elements of the semi-lattice are shown as *programs*. This is done to keep the illustrations comprehensible; however, the reader should keep in mind that the "is-a-slice-of" relation depicted is really a partial order on the programs' *program dependence graphs*.

One reason this formalization is interesting is that it gives a way of expressing $Pre(A, Base, B)$ using lattice-theoretic terminology: in (G, $\wedge$), $Pre(A, Base, B) = A \wedge Base \wedge B$.

Returning to the HPR algorithm, the fact that $\Delta(A, Base)$ and $\Delta(B, Base)$ are combined with $Pre(A, Base, B)$ suggests that this combination is a join operation. The terms $\Delta(A, Base)$ and $\Delta(B, Base)$ themselves suggest that some sort of difference operation exists for elements of the underlying lattice.

What complicates matters is the fact that (G, $\wedge$) is a meet semi-lattice but *not* a lattice (*i.e.*, it has a meet operation but no join operation). In particular, the operation of unioning two dependence graphs (by $\cup_g$), which is used in the HPR algorithm to combine the dependence graphs that represent $\Delta(A, Base)$, $\Delta(B, Base)$, and $Pre(A, Base, B)$, is not a join operation. To see this, consider the union of the dependence graphs for the programs A and B shown below.

A	*B*
program	**program**
$x := 0$;	$x := 0$;
$y := x$;	**if** $w > 2$ **then** $x := 1$ **fi**;
$z := y$	$y := x$
end()	**end**()

The result of $A \cup_g B$ is a dependence graph that corresponds to the program

---

$a \sqcup \top = \top$ $\quad$ $a \sqcap \top = a$
$a \sqcup \bot = a$ $\quad$ $a \sqcap \bot = \bot$

A *meet semi-lattice* (L, $\sqcap$) is defined similarly, but lacks the join operation.

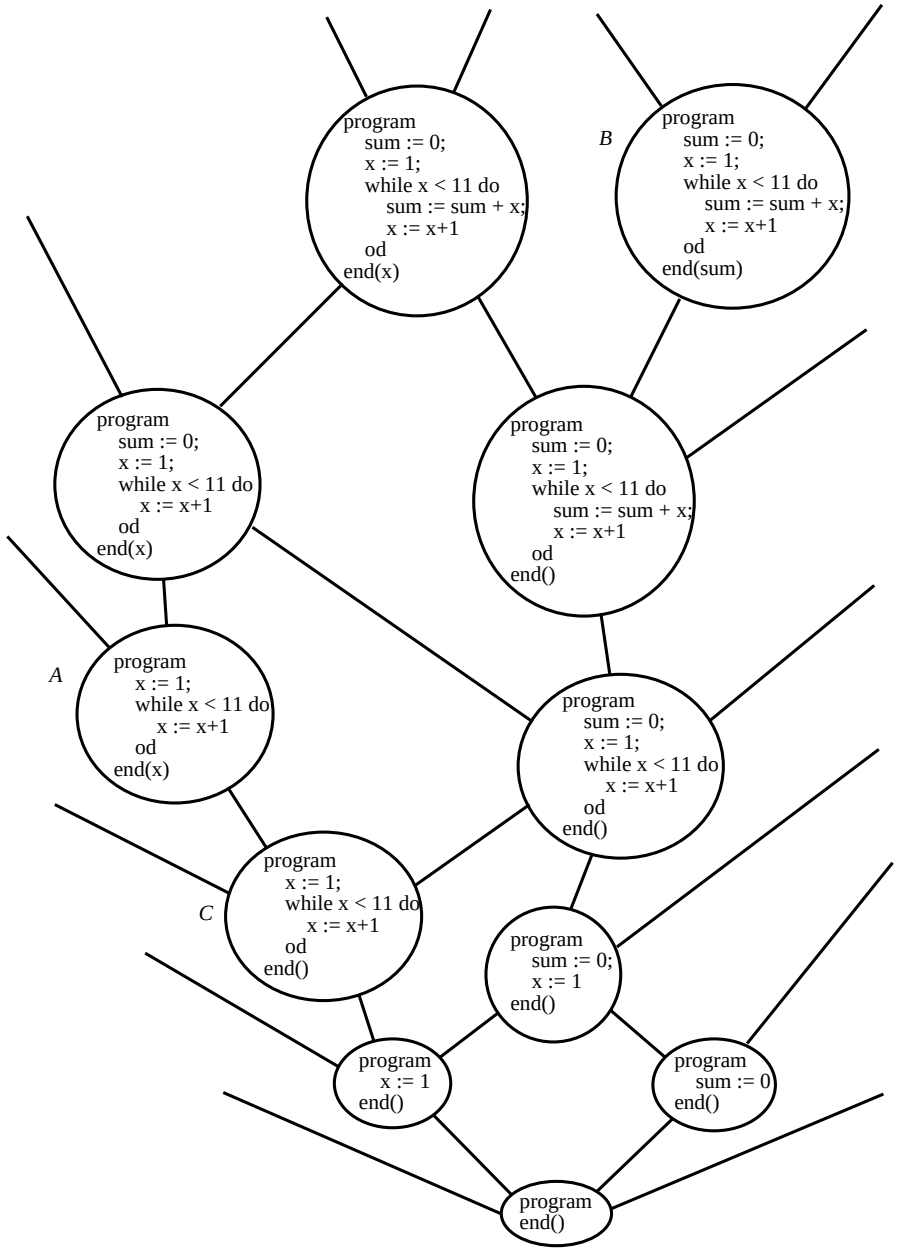


Figure 3. The meet semi-lattice of dependence graphs (G, $\wedge$). The meet operation is illustrated by elements A, B, and C, which are related by $A \wedge B = C$.

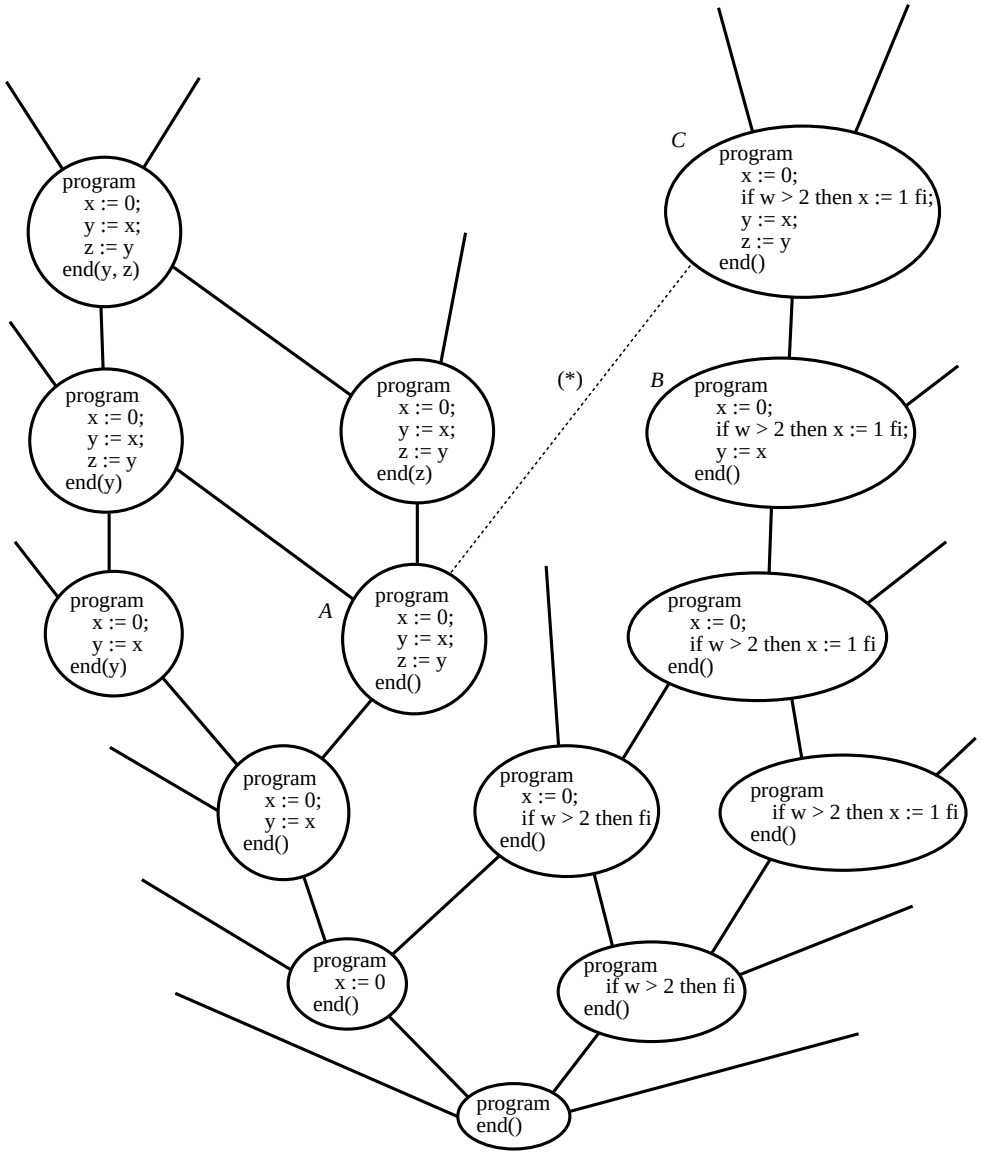


Figure 4. An example illustrating that $(G, \wedge)$ does not have a join operation. There is no element D such that $A \leq D$ and $B \leq D$. In particular, $A \cup_g B = C$ is not such an element. Although $B \leq C$ holds, $A \leq C$ does not hold because the dotted edge marked with (*) is not in the relation "is-a-slice-of".

C

program
  $x := 0$;
  if $w > 2$ **then** $x := 1$ **fi**;
  $y := x$;
  $z := y$
end()

The relationships between A, B, and C in $(G, \wedge)$ are illustrated in Figure 4. For $\cup_g$ (graph union) to be a join operation in $(G, \wedge)$, both A and B must be slices of $A \cup_g B$ (*i.e.*, both $A \leq C$ and $B \leq C$ must hold). However, although both A and B are *subgraphs* of C, only B is a *slice* of C (*i.e.*, $B \leq C$, but $A \not\leq C$). The graph for A is not a slice of C because in C vertex $y := x$ is the target of a flow edge whose source is vertex $x := 1$, whereas there is no such edge incident on vertex $y := x$ in A. Thus, $C/A \neq A$ and so $A \not\leq C$.

In fact, there does not exist *any* element D in $(G, \wedge)$ such that $A \leq D$ and $B \leq D$ both hold. So not only does $\cup_g$ fail to be a join operation, $(G, \wedge)$ has no join operation at all.

3.2. A Brouwerian Algebra of Slice Sets

Rather than work with $(G, \wedge)$, we construct a lattice whose elements consist of sets of slices and perform operations on these sets; in particular, the join operation is set union. However, in order for set intersection to capture common slices, it is necessary to work with sets having a particular structure (rather than arbitrary sets of slices). Similarly, ordinary set difference turns out not to capture the notion of $\Delta(A, Base)$ and $\Delta(B, Base)$ from the HPR algorithm, but a variation on set difference that takes into account the ordering relation on slices can be used instead.

Definition 3.2. A dependence graph $g \in G$ is a *single-point slice* iff there exists a vertex $x \in V(g)$ such that $(g/x) = g$. The symbol G_1 will be used to denote the subset of G consisting of all program dependence graphs that are single-point slices; that is,

$$G_1 \triangleq \{ g \in G \mid \exists x \in V(g) \text{ such that } (g/x) = g \}.$$

Observation. G_1 is a partial order with least element

```
program
end()
```

Example. The partial order of single-point slices is illustrated in Figure 5. Note that the element shown below, which is present in Figure 4 but absent from Figure 5, is *not* a single-point slice, and hence not an element of G_1. For instance, its slice with respect to the final-use vertex for variable x does not include either vertex $sum := 0$ or vertex $sum := sum + x$; on the other hand, its slice with respect to vertex $sum := sum + x$ does not include the final-use vertex for x.

```
program
  sum := 0;
  x := 1;
  while x < 11 do
    sum := sum + x;
    x := x + 1
  end
end(x)
```

Our next construction makes use of the ordering relation on elements of G_1 to create a suitable lattice for expressing program integration.

Definition 3.3. The symbol *DCS* will be used to denote the set of all *downwards-closed sets of single-point slices*; that is,

$$DCS \triangleq \{ S \in P(G_1) \mid \forall x \in G_1 \text{ if } \exists s \in S \text{ such that } x \leq s \text{ then } x \in S \},$$

where $P(G_1)$ denotes the power set of G_1. The notation $DC(p)$, where p is a program, will be used to denote the member of *DCS* that consists of all single-point slices of p; that is,

$$DC(p) \triangleq \{ s \in G_1 \mid s \leq G_p \},$$

where G_p denotes the PDG of p. The notation $DC_g(g)$, where g is a PDG, will be used to denote

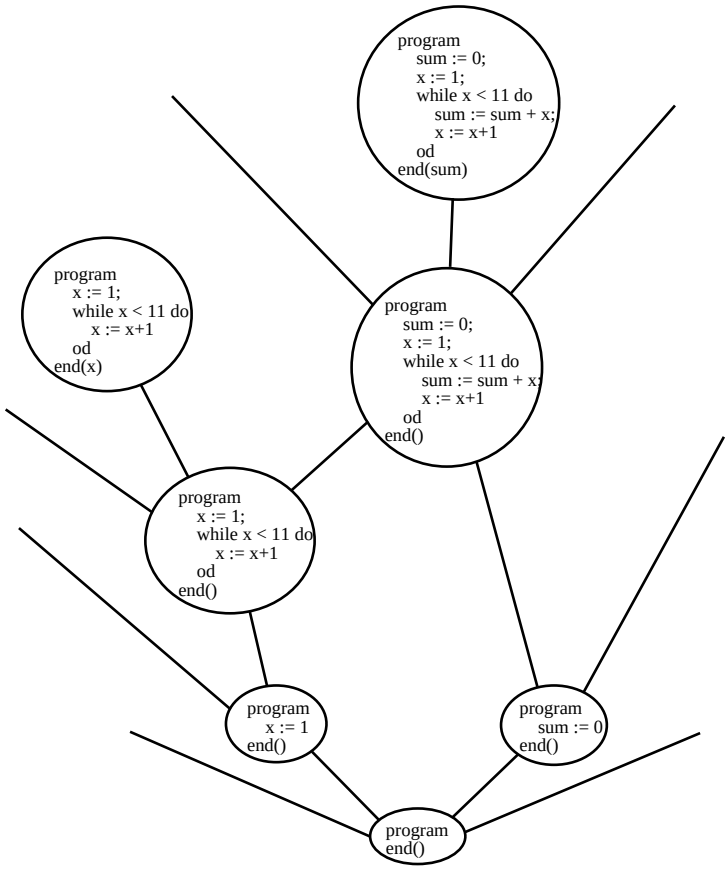


Figure 5. The partial order of single-point slices.

$\{ s \in G_1 \mid s \leq g \}$.

Example.

$$DC\left(\begin{array}{l}\textbf{program}\\ \quad x := 0;\\ \quad y := x;\\ \quad z := y\\ \textbf{end}(x, y, z)\end{array}\right)$$

stands for the set

$$\left\{\begin{array}{lllllll}\textbf{program}, & \textbf{program}, & \textbf{program}, & \textbf{program}, & \textbf{program}, & \textbf{program}, & \textbf{program}\\ \textbf{end}() & x := 0 & x := 0 & x := 0; & x := 0; & x := 0; & x := 0;\\ & \textbf{end}() & \textbf{end}(x) & y := x & y := x & y := x; & y := x;\\ & & & \textbf{end}() & \textbf{end}(y) & z := y & z := y\\ & & & & & \textbf{end}() & \textbf{end}(z)\end{array}\right\}.$$

Example. A second element of *DCS* is shown in Figure 6.

Observation. The algebra (*DCS*, $\cup$, $\cap$), where $\cup$ and $\cap$ denote set union and set intersection, respectively, is a lattice. The partial order on lattice elements is "subset-of" (denoted by $\subseteq$).

Example. Consider again the example programs for which there was no join in (G, $\wedge$). These programs would be represented in (*DCS*, $\cup$, $\cap$) by downwards-closed sets of single-point slices. The set that represents their join is shown in Figure 7.

Definition 3.4. For all $x, y \in DCS$, the *pseudo-difference* between x and y, denoted by $x \dot{-} y$, is defined as

$$x \dot{-} y \triangleq \{ z \in G_1 \mid \exists p \in (x - y) \text{ such that } z \leq p \},$$

where $x - y$ denotes the set difference between x and y. That is, $x \dot{-} y$ is the downwards closure of $x - y$.

The following table illustrates how the pseudo-difference operation ($\dot{-}$) differs from ordinary set difference ($-$):

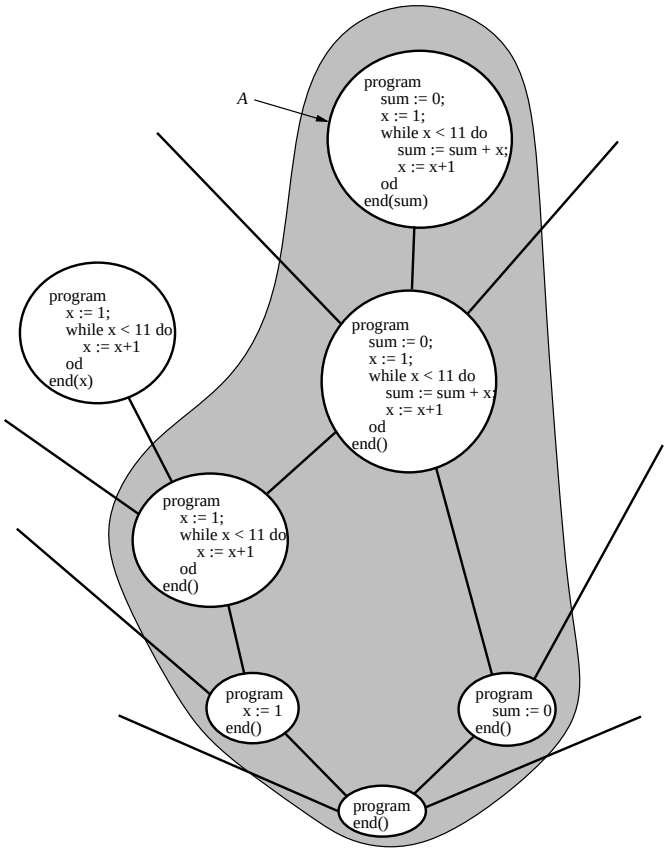


Figure 6. The shaded region indicates the set in *DCS* that represents $DC(A)$.

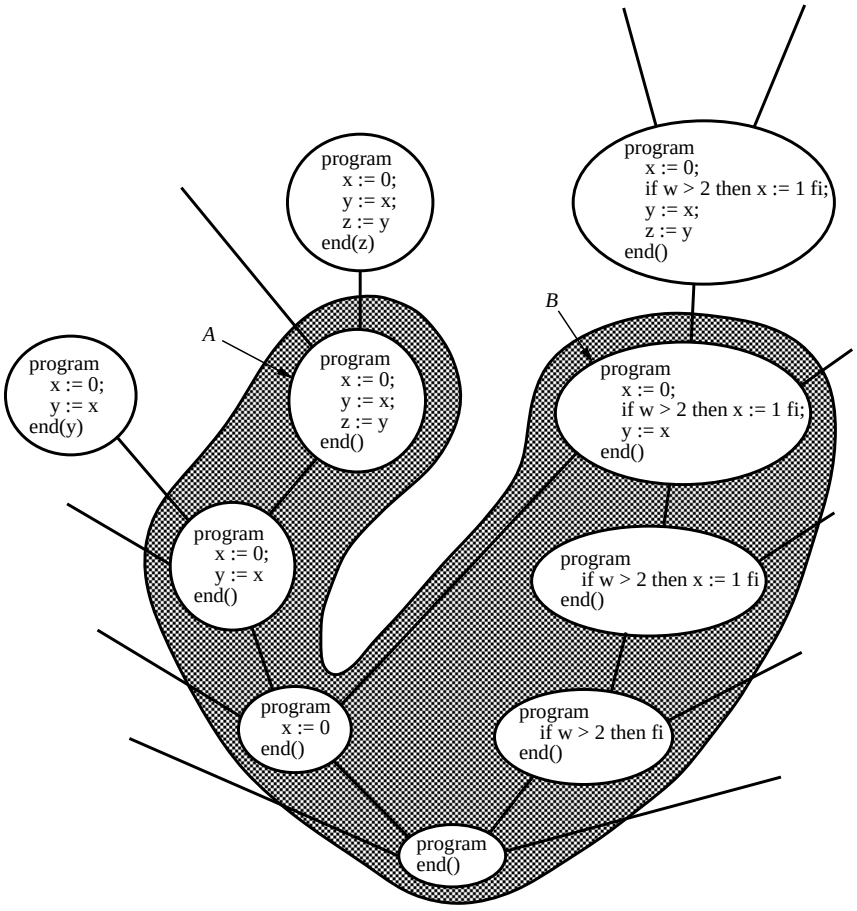


Figure 7. The shaded region indicates the set $DC(A) \cup DC(B)$, which in $(DCS, \cup, \cap)$ is the join of sets $DC(A)$ and $DC(B)$.

a	b	$a - b$	$b - a$
{**program** , **program** **end**() $x := 0$ **end**() }	{**program** , **program** , **program** **end**() $x := 0$ $x := 0;$ **end**() $y := x$ **end**() }	$\varnothing\ (= \perp)$	{**program** $x := 0;$ $y := x$ **end**() }

$a \div b$	$b \div a$
$\varnothing\ (= \perp)$	{**program** , **program** , **program** **end**() $x := 0$ $x := 0;$ **end**() $y := x$ **end**() }

In general, an element of *DCS* can contain multiple maximal elements; these are indicated by the multiple peaks of sets x and y in Figure 8. Figure 8 illustrates the operation of pseudo-difference on two arbitrary elements of *DCS*. The value of $x \div y$ is the downwards closure of $x - y$; thus, $x \div y$ includes both of

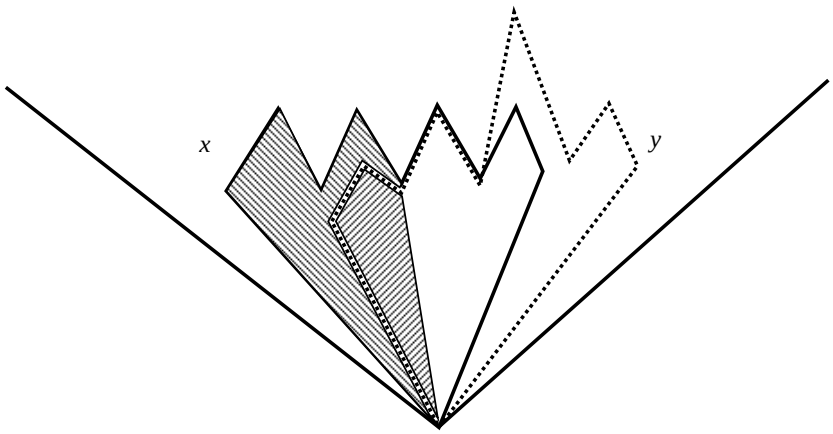


Figure 8. The value of $x \dot{-} y$ is the downwards closure of $x - y$. Thus, $x \dot{-} y$ includes both of the shaded regions.

the shaded regions shown in Figure 8.

The set *DCS*, together with the operations of set union ($\cup$), set intersection ($\cap$), and pseudo-difference ($\dot{-}$), is an instance of what is known as a Brouwerian algebra (defined below). The benefit of this fact is that Brouwerian algebras have a rich set of algebraic laws, consisting of identities and inequalities (see Appendix A). These laws provide a convenient way to establish the integration operation's algebraic properties through simple formula manipulations. (For examples, see Sections 4 and 5 and the appendices.)

Definition 3.5. A *Brouwerian algebra* [16] is an algebra $(L, \sqcup, \sqcap, \dot{-}, \top)$ where

(i) $(L, \sqcup, \sqcap)$ is a lattice with greatest element $\top$.

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

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

It can be shown that L has a least element, given by $\bot = \top \dot{-} \top$, and that $(L, \sqcup, \sqcap)$ is distributive; that is, for all a, b, and c in L,

(iv) $a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$.

(v) $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$.

(For proofs, see [2], page 143-145.)

Remark. We use the symbol $\dot{-}$ to denote the general operation of pseudo-difference in an arbitrary Brouwerian algebra as well as a specific operation in the algebra $(DCS, \cup, \cap, \dot{-}, G_1)$. It should be clear from the context which usage of $\dot{-}$ is intended.

Theorem 3.6. *$(DCS, \cup, \cap, \dot{-}, G_1)$ is a Brouwerian algebra, where $\cup$ is set union and $\cap$ is set intersection.*

Proof. Because the elements of *DCS* are downwards-closed sets of single-point slices, it is clear that G_1, which consists of all single-point slices, is a superset of any element of *DCS*. Suppose $s \in G_1$ and x is a single-point slice of s; because x—being a single-point slice—must also be a member of G_1, it follows that G_1 is itself downwards closed (*i.e.*, $G_1 \in DCS$). *DCS* is closed under $\cup$ and $\cap$, and $(DCS, \cup, \cap)$ is a lattice ordered by set inclusion.

It remains to be shown that $\dot{-}$ has the properties required of a pseudo-difference; that is, we must show (1) *DCS* is closed under $\dot{-}$ and (2) for all $a, b, c \in DCS$, $a \dot{-} b \subseteq c$ iff $a \subseteq b \cup c$.

To show property (1), consider any two elements $a, b \in DCS$. From Definition 3.4, we know that $a \dot{-} b$ is the downwards closure (under the "is-a-single-point-slice-of" relation) of $a - b$, and hence $a \dot{-} b \in DCS$. (Note that $a - b$ denotes the set difference of a and b; $a - b$ is not necessarily downwards closed, and hence, in general, is not a member of *DCS*.)

Two show property (2), there are two cases to consider.

$\Rightarrow$ *case: Assuming $a \dot{-} b \subseteq c$, we must show that $a \subseteq b \cup c$.*

From Definition 3.4, we know that $a - b \subseteq a \dot{-} b$.

$a - b \subseteq c$	by transitivity
$(a - b) \cup b \subseteq b \cup c$	
$a \cup b \subseteq b \cup c$	because $(a - b) \cup b = a \cup b$
$a \subseteq b \cup c$	because $a \subseteq a \cup b$

$\Leftarrow$ *case: Assuming $a \subseteq b \cup c$, we must show that $a \dot{-} b \subseteq c$.*

Let z be a member of $a \dot{-} b$; we will show that $z \in c$. From Definition 3.4 we know that there exists a (single-point slice) $p \in a$ such that $p \notin b$ and $z \leq p$. By the downwards-closure property of elements of *DCS*, because $p \in a$ we know that $z \in a$ as well. There are two cases to consider:

(i) Suppose $z \notin b$. By the assumption that $a \subseteq b \cup c$ and the fact that $z \in a$, we have $z \in b \cup c$. However, because $z \notin b$, we conclude that $z \in c$.

(ii) Suppose $z \in b$. Consider again the element p, where $p \in a$, $p \notin b$, and $z \leq p$. By the assumption that $a \subseteq b \cup c$ and the fact that $p \in a$, we have $p \in b \cup c$. However, because $p \notin b$, we have $p \in c$. But because $z \leq p$ and because c—being an element of *DCS*—is downwards closed, we conclude that $z \in c$.

□

3.3. Relationship Between Brouwerian and Boolean Algebras

A Brouwerian algebra is similar, but not identical, to a Boolean algebra. The relationship between Boolean and Brouwerian algebras can be characterized as follows [16]: for all elements a, define the *Brouwerian complement* by $\neg a \triangleq \top \dot{-} a$; a Brouwerian algebra is Boolean iff $\neg \neg a = a$.

Example. It may be helpful to keep in mind the following example of a Brouwerian algebra that is not a Boolean algebra. The elements are the non-negative integers (together with a top element $\top$); the ordering is arithmetic order (*i.e.*, $0 \sqsubset 1 \sqsubset 2 \ldots \sqsubset \top$).

The operations $\sqcup$, $\sqcap$, and $\dot{-}$ are defined as follows:

$$a \sqcup b \triangleq \max(a, b)$$
$$a \sqcap b \triangleq \min(a, b)$$
$$a \dot{-} b \triangleq \begin{cases} 0 & \text{if } a \sqsubseteq b \\ a & \text{if } a \sqsupset b \end{cases}$$

Note that $a \dot{-} b$ exhibits the required property, namely that $a \dot{-} b \sqsubseteq c$ iff $a \sqsubseteq \max(b, c)$. In particular, to show that $a \dot{-} b \sqsubseteq c$ implies $a \sqsubseteq \max(b, c)$, we observe that either $a \sqsubseteq b$, in which case $a \sqsubseteq \max(b, c)$ follows immediately, or else $a \sqsupset b$, in which case $a = a \dot{-} b \sqsubseteq c$, so again $a \sqsubseteq \max(b, c)$. Conversely, to show that $a \sqsubseteq \max(b, c)$ implies $a \dot{-} b \sqsubseteq c$, we observe that either $b \sqsubseteq c$, in which case $c = \max(b, c) \sqsupseteq a \sqsupseteq a \dot{-} b$ (*i.e.*, $c \sqsupseteq a \dot{-} b$, as was to be shown), or else $b \sqsupset c$, in which case $b = \max(b, c) \sqsupseteq a$, so $a \dot{-} b = 0 \sqsubseteq c$.

To show that this algebra is not Boolean, we must show that there is some a for which $\neg \neg a \neq a$. In fact, this holds for all a such that $0 \sqsubset a \sqsubset \top$, as shown in the following table:

a	$\neg a$	$\neg \neg a$
0	$\top$	0
$0 \sqsubset a \sqsubset \top$	$\top$	0 ($\neq a$)
$\top$	0	$\top$

Because Boolean algebras are Brouwerian algebras, but not vice versa, some of the properties that hold in Boolean algebras do not hold in Brouwerian algebras. (Consequently, applying one's intuition about Boolean algebras to Brouwerian algebras can be risky.) For example, the laws for distributing $\dot{-}$ over $\sqcup$ and $\sqcap$ in Brouwerian algebra are somewhat different from the laws for distributing – over $\sqcup$ and $\sqcap$ in Boolean algebra. Two of the laws are the same:

Proposition A.14. $(b \dot{-} a) \sqcup (c \dot{-} a) = (b \sqcup c) \dot{-} a$.[8]

Proposition A.15. $(c \dot{-} a) \sqcup (c \dot{-} b) = c \dot{-} (a \sqcap b)$.

[8]The basic algebraic laws that hold for elements of Brouwerian algebras are presented in Appendix A.

However, the laws for distributing $\div$ through $\sqcap$ on the left and $\sqcup$ on the right are weaker:[9]

Proposition A.26. $(a \sqcap b) \div c \sqsubseteq (a \div c) \sqcap (b \div c)$.

Proposition A.29. $c \div (a \sqcup b) \sqsubseteq (c \div a) \sqcap (c \div b)$.

We can show by means of examples that the inequalities in Propositions A.26 and A.29 are, at times, strict.

a	b	c	$(a \sqcap b) \div c$	$a \div c$	$b \div c$	$(a \div c) \sqcap (b \div c)$
DC ⌈**program** $x := 0;$ $y := x$ ⌊**end**() ⌋	DC ⌈**program** $x := 0;$ $z := x$ ⌊**end**() ⌋	DC ⌈**program** $x := 0$ ⌊**end**() ⌋	$\bot$	DC ⌈**program** $x := 0;$ $y := x$ ⌊**end**() ⌋	DC ⌈**program** $x := 0;$ $z := x$ ⌊**end**() ⌋	DC ⌈**program** $x := 0$ ⌊**end**() ⌋

In this example $(a \sqcap b) \div c$ is strictly less than $(a \div c) \sqcap (b \div c)$ because the following two slices occur in both $a \div c$ and $b \div c$:

program
end()

program
$x := 0$
end().

a	b	c	$c \div (a \sqcup b)$	$c \div a$	$c \div b$	$(c \div a) \sqcap (c \div b)$
DC ⌈**program** $x := 0;$ $y := x$ ⌊**end**() ⌋	DC ⌈**program** $x := 0;$ $z := x$ ⌊**end**() ⌋	DC ⌈**program** $x := 0;$ $y := x;$ $z := x$ ⌊**end**() ⌋	$\bot$	DC ⌈**program** $x := 0;$ $z := x$ ⌊**end**() ⌋	DC ⌈**program** $x := 0;$ $y := x$ ⌊**end**() ⌋	DC ⌈**program** $x := 0$ ⌊**end**() ⌋

In this example $c \div (a \sqcup b)$ is strictly less than $(c \div a) \sqcap (c \div b)$ because the following two slices occur in both $c \div a$ and $c \div b$:

program
end()

program
$x := 0$
end().

3.4. Integration of Elements of a Brouwerian Algebra

We now introduce a ternary operation on elements of a Brouwerian algebra that, for the algebra of downwards-closed sets of single-point slices (DCS, $\cup$, $\cap$, $\div$, G_1) discussed in Section 3.2, corresponds very closely to the HPR algorithm. The integration operation on elements of a Brouwerian algebra, denoted by $a\,[base\,]b$, combines two elements a and b with respect to a third element *base*.

Definition 3.7. The *integration* of elements a and b with respect to *base*, denoted by $a\,[base\,]b$, is defined as

$$a\,[base\,]b \triangleq (a \div base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \div base).$$

If $a\,[base\,]b = m$, we refer to element *base* as the *base*, elements a and b as the *integrands*, and element m as the *result* of the integration.

---

[9] Note, however, that Proposition A.16 does provide an identity that can be used to transform $c \div (a \sqcup b)$:

Proposition A.16. $(c \div b) \div a = c \div (a \sqcup b) = (c \div a) \div b$.

The integration operation $a\,[base\,]b$ in $(DCS,\ \cup,\ \cap,\ \div,\ G_1)$ provides a method for integrating programs that is an alternative to the HPR algorithm.

Example. The table shown in Figure 9, which indicates what slices are members of the sets $DC(A)$, $DC(Base)$, $DC(B)$, $DC(A) \div DC(Base)$, $DC(A) \cap DC(Base) \cap DC(B)$, $DC(B) \div DC(Base)$, and $DC(A)[DC(Base)]DC(B)$, illustrates the integration operation (in the algebra of downwards-closed sets of single-point slices) for the same example used to illustrate the HPR algorithm in Section 2.2. This produces a result equivalent to the one obtained in Section 2.2; the set of slices computed for $DC(A)[DC(Base)]DC(B)$ (shown in the last line of the table given in Figure 9) corresponds to the program

program
  $x := 0$;
  $y := x$;
  $z := y$
end(x, z).

3.5. Relationship to the HPR Algorithm

The integration operation in $(DCS,\ \cup,\ \cap,\ \div,\ G_1)$ and the HPR algorithm are related methods for integrating programs, but differ substantially in detail. For example, to integrate programs *a* and *b* with respect to *base*, the HPR algorithm performs the operation

$$\Delta(a,\ base)\ \cup_g\ Pre\,(a,\ base,\ b)\ \cup_g\ \Delta(b,\ base),$$

which manipulates three *individual* program dependence graphs. By contrast, the integration operation in $(DCS,\ \cup,\ \cap,\ \div,\ G_1)$ manipulates three *sets* of program dependence graphs. To integrate programs *a* and *b* with respect to *base*, the following operation is performed:

$$(DC(a) \div DC(base)) \cup (DC(a) \cap DC(base) \cap DC(b)) \cup (DC(b) \div DC(base)).$$

Despite the fact that the two algorithms perform different operations, their final results are related, as

---

Term	Slice						
	program **end**()	**program** $x := 0$ **end**()	**program** $x := 0$ **end**(x)	**program** $x := 0$; $y := x$ **end**()	**program** $x := 0$; $y := x$ **end**(y)	**program** $x := 0$; $y := x$; $z := y$ **end**()	**program** $x := 0$; $y := x$; $z := y$ **end**(z)
$DC(A)$	×	×	×				
$DC(Base)$	×	×	×	×	×		
$DC(B)$	×	×	×	×	×	×	×
$DC(A) \div DC(Base)$							
$DC(A) \cap DC(Base) \cap DC(B)$	×	×	×				
$DC(B) \div DC(Base)$	×	×		×		×	×
$DC(A)[DC(Base)]DC(B)$	×	×	×	×		×	×

Figure 9. The table given above illustrates the integration operation in the algebra of downwards-closed sets of single-point slices for the same example used to illustrate the HPR algorithm in Section 2.2. The last line of the table indicates which slices are members of the set $DC(A)[DC(Base)]DC(B)$.

explained below.

First, consider the graph-manipulation operations Δ and *Pre* used in the HPR algorithm. Δ is related to $\div$ as follows:

$$DC_g(\Delta(a, base)) = DC(a) \div DC(base).$$

Pre is related to $\cap$ as follows:

$$DC_g(Pre(a, base, b)) = DC(a) \cap DC(base) \cap DC(b).$$

Second, consider the test for Type I interference in the HPR algorithm. The test is based on a comparison of slices of the merged graph *m* with slices of the graphs for programs *a* and *b*. The slices $\Delta(a, base)$ and $\Delta(b, base)$, respectively, represent the potentially changed computations of integrands *a* and *b* with respect to *base*. There is Type I interference if graph *m* does not preserve these slices, that is, if either $\Delta(a, base)$ or $\Delta(b, base)$ is not a slice of *m*. Thus, the test for Type I interference can be expressed as follows: There is Type I interference iff

$$\Delta(a, base) \not\leq \Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)$$

or

$$\Delta(b, base) \not\leq \Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base).^{10}$$

The reason for the Type I interference test in the HPR algorithm is that the operation of graph union ($\cup_g$) can "corrupt" slices; that is, it can create a graph whose slices are not slices of either argument. For instance, consider again programs *A* and *B* from Figure 4. The result of $A \cup_g B$ is a dependence graph that corresponds to the program *C* shown below.

```
                C
program
   x := 0;
   if w > 2 then x := 1 fi;
   y := x;
   z := y
end()
```

The slice of *C* with respect to statement $z := y$ (which yields the entire program *C*) is not a slice of either *A* or *B*.

From these observations—together with the definition of the integration operation in $(DCS, \cup, \cap, \div, G_1)$—we conclude that if the HPR algorithm does not report Type I interference, the two integration methods compute answers that correspond. In particular, there is a way of converting the PDG created by the HPR algorithm into the set of single-point slices computed by the integration operation in $(DCS, \cup, \cap, \div, G_1)$. This is captured by the following proposition:

Proposition 3.8. *If the integration of programs a and b with respect to base via the HPR algorithm passes the Type I interference test, then*

$$DC_g(\Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)) = DC(a)[DC(base)]DC(b).$$

---

[10] It is natural to ask why the interference test does not have a third clause to test whether

$$Pre(a, base, b) \not\leq \Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base).$$

This test is absent because, as shown in [22, 25], it is unnecessary: $Pre(a, base, b)$ is *always* a slice of $\Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)$.

Recall from Section 2.2 that Type I interference is not the only way in which the merged dependence graph *m* created by the HPR algorithm can fail to represent a satisfactory integrated program. The final step of the HPR algorithm involves reconstituting a program from dependence graph *m*; however, if there is no program whose dependence graph is *m*, there is Type II interference (and graph *m* is said to be *infeasible*).

So far we have not discussed the notion of interference in connection with the integration operation in (DCS, $\cup$, $\cap$, $\dot{-}$, G_1). The notion of interference is slightly different from the HPR algorithm; in particular, because the $\cup$, $\cap$, and $\dot{-}$ operations in (DCS, $\cup$, $\cap$, $\dot{-}$, G_1) can never "corrupt" slices, there is only *one* notion of interference for integration in (DCS, $\cup$, $\cap$, $\dot{-}$, G_1), namely, infeasibility.

Definition 3.9. A slice set $s \in DCS$ is *feasible* iff there exists a program p such that $DC(p) = s$. If no such program exists, s is *infeasible*.

Thus, if the goal is to integrate programs *a* and *b* with respect to *base* using the integration operation in (DCS, $\cup$, $\cap$, $\dot{-}$, G_1), integrands *a* and *b* interfere if there is no program that corresponds to the slice set $DC(a)[DC(base)]DC(b)$.

We now show that $DC(a)[DC(base)]DC(b)$ is infeasible iff either Type I or Type II interference is reported by the HPR algorithm (and thus the class of integrations handled successfully by the integration operation in (DCS, $\cup$, $\cap$, $\dot{-}$, G_1) coincides with the class handled successfully by the HPR algorithm). Stated in the contrapositive, we have:

Proposition 3.10. *$DC(a)[DC(base)]DC(b)$ is feasible iff the integration of a and b with respect to base by the HPR algorithm succeeds (i.e., neither Type I nor Type II interference is reported).*

Proof.

$\Leftarrow$ *case: Assuming that the integration of a and b with respect to base by the HPR algorithm succeeds, we must show that $DC(a)[DC(base)]DC(b)$ is feasible.*

By the assumption that the HPR algorithm does not report Type II interference, we know that $\Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)$ is feasible. Thus, there exists a program P (with PDG G_P) such that $G_P = \Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)$. Consequently,

$$\begin{aligned} DC(P) &= DC_g(G_P) \\ &= DC_g(\Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)) \\ &= DC(a)[DC(base)]DC(b) \end{aligned}$$

By the assumption that Type I interference is not reported and Proposition 3.8

Hence, $DC(a)[DC(base)]DC(b)$ is feasible.

$\Rightarrow$ *case: Assuming that $DC(a)[DC(base)]DC(b)$ is feasible, we must show that the integration of a and b with respect to base by the HPR algorithm succeeds.*

By assumption, there exists a program P (with PDG G_P) such that $DC_g(G_P) = DC(a)[DC(base)]DC(b)$.

Note that for any PDG H, $H = \bigcup_{s \in DC_g(H)}^{g} s$. We use this observation in the following derivation:

$$
\begin{aligned}
G_P &= \bigcup_{s \in DC_g(G_P)}{}_g\ s. \\
&= \bigcup_{s \in DC(a)[DC(base)]DC(b)}{}_g\ s \\
&= \bigcup_{s \in DC(a) \div DC(base)}{}_g\ s \ \cup_g \bigcup_{s \in DC(a) \cap DC(base) \cap DC(b)}{}_g\ s \ \cup_g \bigcup_{s \in DC(b) \div DC(base)}{}_g\ s \\
&= \bigcup_{s \in DC_g(\Delta(a,\, base))}{}_g\ s \ \cup_g \bigcup_{s \in DC_g(Pre(a,\, base,\, b))}{}_g\ s \ \cup_g \bigcup_{s \in DC_g(\Delta(b,\, base))}{}_g\ s \\
&= \Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base) \qquad (*)
\end{aligned}
$$

There are now two cases to consider:

(i) Suppose the HPR algorithm reports Type I interference.

Without loss of generality, assume that $\Delta(a, base) \not\leq \Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)$. Thus, we have

$$
\begin{aligned}
DC_g(\Delta(a, base)) &\not\subseteq DC_g(\Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)) \\
&= DC_g(G_P). \qquad \text{by (*)}
\end{aligned}
$$

By the definition of the integration operation in $(DCS, \cup, \cap, \div, G_1)$,

$$DC(a) \div DC(base) \subseteq DC(a)[DC(base)]DC(b).$$

However, this leads to the following contradiction:

$$DC(a) \div DC(base) = DC_g(\Delta(a, base)) \not\subseteq DC_g(G_P) = DC(a)[DC(base)]DC(b).$$

Thus, the HPR algorithm cannot report Type I interference.

(ii) Suppose the HPR algorithm reports Type II interference.

This leads to an immediate contradiction since, by (*),

$$G_P = \Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)$$

and hence $\Delta(a, base) \cup_g Pre(a, base, b) \cup_g \Delta(b, base)$ is feasible.

Consequently, neither Type I nor Type II interference arises from the integration of *a* and *b* with respect to *base* by the HPR algorithm. □

Although Proposition 3.10 shows that the class of integrations handled successfully by the integration operation in $(DCS, \cup, \cap, \div, G_1)$ coincides with the class handled successfully by the HPR algorithm, there is a difference in how the two algorithms handle *unsuccessful* integrations of the kind where the HPR algorithm reports Type I interference. By Proposition 3.10, when the HPR algorithm reports Type I interference the slice set resulting from the integration operation in $(DCS, \cup, \cap, \div, G_1)$ is infeasible; however, the slices in $DC(a)[DC(base)]DC(b)$ are "uncorrupted," which gives the integration operation in $(DCS, \cup, \cap, \div, G_1)$ certain advantages over the HPR algorithm. For example, suppose we perform a succession of integrations to propagate a change through the development history of a program. With the HPR algorithm, once Type I interference is detected it is not meaningful to perform any subsequent integrations because Type I interference indicates that the resulting graph contains slices not found in any of the argument graphs. By contrast, with the integration operation in $(DCS, \cup, \cap, \div, G_1)$ although an intermediate value may be infeasible—indicating interference—none of the slices in the slice set have been "corrupted." If a subsequent integration involving infeasible elements produces a feasible result, we are guaranteed that the result contains only slices that were found in the argument sets.

The algebraic identities and inequalities we use to establish the integration operation's algebraic properties hold in all Brouwerian algebras, not just $(DCS, \cup, \cap, \div, G_1)$, and therefore the results we obtain hold in all Brouwerian algebras as well. It is important to understand that the notion of interference is not part of this algebraic framework. For example, the classification of the elements of *DCS* into feasible and

infeasible elements is an issue that is specific to *DCS*; the feasibility issue is orthogonal to the algebraic structure of (*DCS*, $\cup$, $\cap$, $\dot{-}$, G_1): the infeasible elements of *DCS* are subject to the same algebraic laws as the feasible elements. Thus, in the remainder of the paper the issue of interference is ignored; as with the algebra (*DCS*, $\cup$, $\cap$, $\dot{-}$, G_1), each application of the integration framework will have an associated notion of interference.

One of the virtues of the HPR algorithm is that there is a theorem, the Integration Theorem (Theorem 2.9), that characterizes the execution behavior of the program produced by a successful integration (*i.e.*, one for which there is neither Type I nor Type II interference) in terms of the execution behaviors of the base program and the two integrands. This same sort of result also holds for the integration operation in (*DCS*, $\cup$, $\cap$, $\dot{-}$, G_1). By Proposition 3.10, $DC(a)[DC(base)]DC(b)$ is feasible iff the HPR algorithm succeeds. By Proposition 3.8, it consists of exactly the single-point slices of the merged graph created by the HPR algorithm. Because the Integration Theorem holds for the result of integrating by the HPR algorithm, whenever $DC(a)[DC(base)]DC(b)$ is feasible the Integration Theorem also holds for any of the programs that correspond to $DC(a)[DC(base)]DC(b)$.

To summarize, the integration operation in (*DCS*, $\cup$, $\cap$, $\dot{-}$, G_1) generalizes the HPR algorithm but preserves its most important property, namely that the execution behavior of the integrated program meets the semantic criterion of the integration model that was introduced in Section 1.1.

4. Algebraic Properties of the Integration Operation

Because the integration operation is defined solely in terms of $\sqcup$, $\sqcap$, and $\dot{-}$, it has an analogue in all Brouwerian algebras, not just the algebra (*DCS*, $\cup$, $\cap$, $\dot{-}$, G_1) from Section 3.2. Because we will study the integration operation's properties strictly from an algebraic standpoint, our results apply to this operation in *all* Brouwerian algebras.

4.1. Basic Algebraic Properties of the Integration Operation

It is not difficult to show that the following basic properties hold for the integration operation:

$a[a]b = b$	$a[base]a = a$
$a[\bot]b = a \sqcup b$	$a[\top]b = a \sqcap b$
$a[a \sqcap b]b = a \sqcup b$	$a[a \sqcup b]b = a \sqcap b$
$a[base]\bot = a \dot{-} base$	$a[base]\top = a \sqcup (\top \dot{-} base)$
$a[base](x_1 \sqcup x_2) = a[base]x_1 \sqcup a[base]x_2$	$a[base](x_1 \sqcap x_2) \sqsubseteq a[base]x_1 \sqcap a[base]x_2$
$a[base]b$ is monotonic in a	$a[base]b$ is antimonotonic in $base$
$a[x_1 \sqcup x_2]b \sqsubseteq a[x_1]b \sqcap a[x_2]b$	$a[x_1 \sqcap x_2]b = a[x_1]b \sqcup a[x_2]b$
$a[base]b \dot{-} base = (a \dot{-} base) \sqcup (b \dot{-} base)$	$a[base]b \dot{-} b = (a \dot{-} base) \dot{-} b$

Proofs of these properties are given in Appendix B.

We now show that the integration operation can be slightly simplified to

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

That is, we can show that

$$(a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base) = (a \dot{-} base) \sqcup (a \sqcap b) \sqcup (b \dot{-} base).$$

Lemma 4.1. $(a \dot{-} base) \sqcup (a \sqcap base \sqcap b) = (a \dot{-} base) \sqcup (a \sqcap b)$.

Proof.

$$
\begin{aligned}
(a \div base) \sqcup (a \sqcap base \sqcap b) &= ((a \div base) \sqcup (a \sqcap base)) \sqcap ((a \div base) \sqcup b) \\
&= a \sqcap ((a \div base) \sqcup b) && \text{by Proposition A.23}[11] \\
&= (a \sqcap (a \div base)) \sqcup (a \sqcap b) \\
&= (a \div base) \sqcup (a \sqcap b) && \text{by Proposition A.11}
\end{aligned}
$$

□

Corollary 4.2. $a\,[base\,]b = (a \div base) \sqcup (a \sqcap b) \sqcup (b \div base)$.

Proof. Immediate from the definition of $a\,[base\,]b$ and Lemma 4.1. □

4.2. Associativity of Integration

The algebraic properties of the integration operation are of particular interest when dealing with compositions of integrations. For example, if three variants of a given base program are to be integrated by a pair of (two-variant) integrations, it is important to know whether there is a law of associativity to guarantee that it does not matter which two variants are integrated first. In this section, we prove that such a law of associativity does hold for the integration operation. We also generalize the integration operation to simultaneously integrate more than two variants with a given base element, and show that a three-variant simultaneous integration can be done as a succession of two-variant integrations.

Definition 4.3. The *simultaneous integration* of elements $x_1, x_2, \cdots, x_n$ with respect to element *base*, denoted by $(x_1\,[base\,]x_2, \cdots, x_n)$, is defined as
$(x_1\,[base\,]x_2, \cdots, x_n) \triangleq (x_1 \div base) \sqcup (x_2 \div base) \sqcup \cdots \sqcup (x_n \div base) \sqcup (x_1 \sqcap x_2 \sqcap \cdots \sqcap x_n \sqcap base)$.

Theorem 4.4. (Generalized Associativity Theorem). $(x\,[base\,]y)[base\,]z = x\,[base\,](y\,[base\,]z)$
$= (x\,[base\,]z)[base\,]y = (x\,[base\,]y)[base\,](x\,[base\,]z) = (x\,[base\,]y)[x\,](x\,[base\,]z) = (x\,[base\,]y, z)$.

Theorem 4.4, which relates six different ways of integrating three variants with respect to a given base element, is illustrated in Figure 10.

Because the integration operator is a ternary operator, Theorem 4.4 is a generalization of the ordinary kind of associative law for binary operators, namely $(x \otimes y) \otimes z = x \otimes (y \otimes z)$. We can simplify the generalized law to the ordinary law by currying the integration operator _ [_]_ with respect to its middle argument, in this case *base*, which gives us a binary operator _ [*base*]_. The first two clauses of Theorem 4.4 are of the form $(x \otimes y) \otimes z = x \otimes (y \otimes z)$, with operator _ ⊗ _ replaced by _ [*base*]_.

Proof of Theorem 4.4.

Part I. Show that $(x\,[base\,]y)[base\,]z = x\,[base\,](y\,[base\,]z) = (x\,[base\,]z)[base\,]y = (x\,[base\,]y, z)$.

$$
\begin{aligned}
&(x\,[base\,]y)[base\,]z \\
&\quad = (x\,[base\,]y \div base) \sqcup (x\,[base\,]y \sqcap base \sqcap z) \sqcup (z \div base) \\
&\quad = (x \div base) \sqcup (y \div base) \sqcup (((x \div base) \sqcup (x \sqcap base \sqcap y) \sqcup (y \div base)) \sqcap base \sqcap z) \sqcup (z \div base) \\
&\quad = (x \div base) \sqcup (y \div base) \sqcup (z \div base) \sqcup (x \sqcap y \sqcap z \sqcap base) \\
&\quad = (x\,[base\,]y, z)
\end{aligned}
$$

By analogous arguments, one can show $x\,[base\,](y\,[base\,]z) = (x\,[base\,]z)[base\,]y = (x\,[base\,]y, z)$.

Part II. Show that $(x\,[base\,]y)[base\,](x\,[base\,]z) = (x\,[base\,]y, z)$.

---

[11]The laws used to justify proof steps are listed in Appendixes A, B, and C.

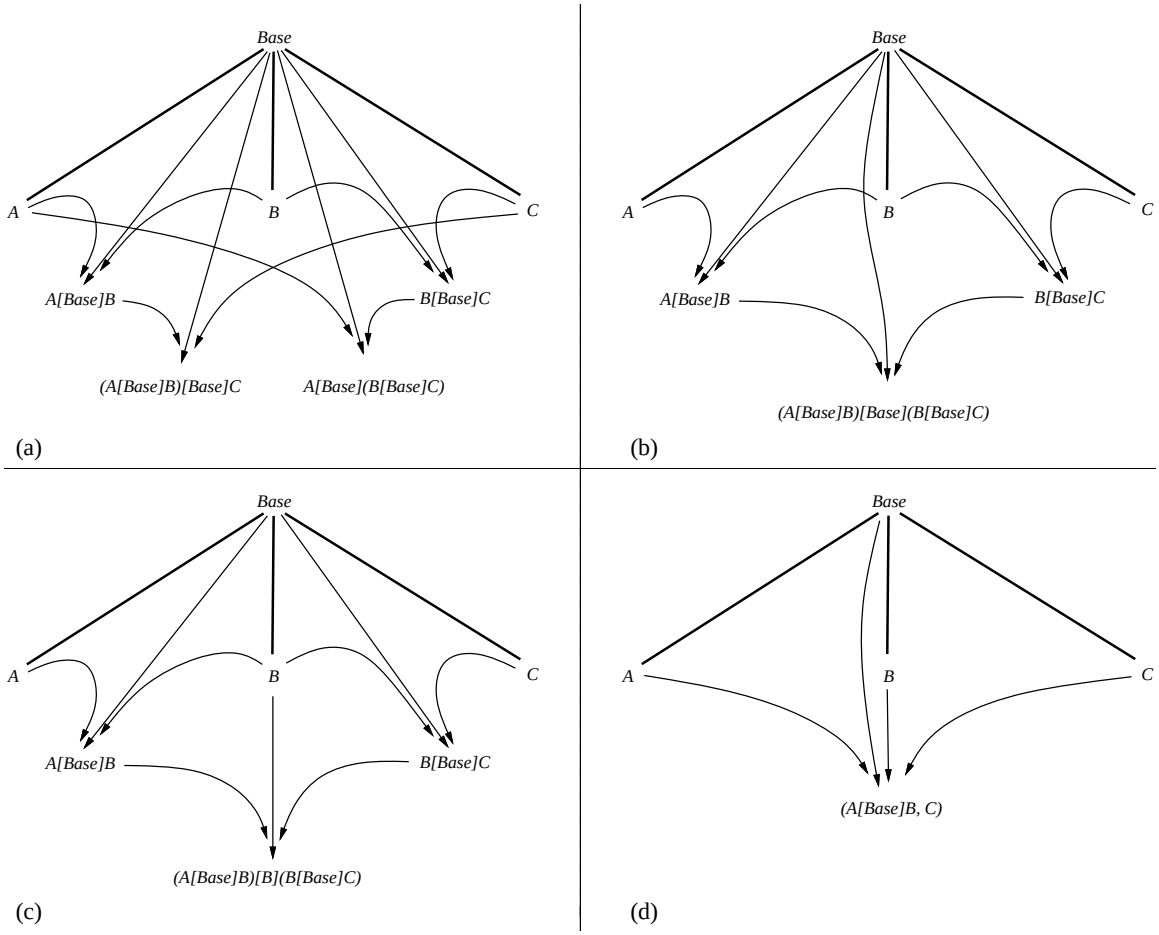


Figure 10. Associativity properties of the integration operation. By the Generalized Associativity Theorem (Theorem 4.4) all the computations illustrated above produce the same answer.

$$
\begin{aligned}
&(x\,[base\,]y)[base\,](x\,[base\,]z)\\
&\quad = (x\,[base\,]y \dot{-} base) \sqcup (x\,[base\,]y \sqcap base \sqcap x\,[base\,]z) \sqcup (x\,[base\,]z \dot{-} base)\\
&\quad = (x \dot{-} base) \sqcup (y \dot{-} base)\\
&\qquad \sqcup (((x \dot{-} base) \sqcup (x \sqcap base \sqcap y) \sqcup (y \dot{-} base)) \sqcap base \sqcap ((x \dot{-} base) \sqcup (x \sqcap base \sqcap z) \sqcup (z \dot{-} base)))\\
&\qquad \sqcup (x \dot{-} base) \sqcup (z \dot{-} base)\\
&\quad = (x \dot{-} base) \sqcup (y \dot{-} base) \sqcup (z \dot{-} base) \sqcup (x \sqcap y \sqcap z \sqcap base)\\
&\quad = (x\,[base\,]y,\, z)
\end{aligned}
$$

Part III. Show that $(x\,[base\,]y)[x\,](x\,[base\,]z) = (x\,[base\,]y,\, z)$.

$$
\begin{aligned}
&(x\,[base\,]y)[x\,](x\,[base\,]z)\\
&\quad = (x\,[base\,]y \dot{-} x) \sqcup (x\,[base\,]y \sqcap x \sqcap x\,[base\,]z) \sqcup (x\,[base\,]z \dot{-} x)\\
&\quad = ((y \dot{-} base) \dot{-} x)\\
&\qquad \sqcup (((x \dot{-} base) \sqcup (x \sqcap base \sqcap y) \sqcup (y \dot{-} base)) \sqcap x \sqcap ((x \dot{-} base) \sqcup (x \sqcap base \sqcap z) \sqcup (z \dot{-} base)))\\
&\qquad \sqcup ((z \dot{-} base) \dot{-} x)\\
&\quad = ((y \dot{-} base) \dot{-} x)
\end{aligned}
$$

$$\begin{aligned}
&\sqcup((x \dot{-} base) \sqcap x) \sqcup ((x \dot{-} base) \sqcap (x \sqcap base \sqcap z)) \sqcup ((x \dot{-} base) \sqcap (z \dot{-} base)) \\
&\quad \sqcup((x \dot{-} base) \sqcap base \sqcap y) \sqcup (x \sqcap base \sqcap z \sqcap y) \sqcup (x \sqcap base \sqcap y \sqcap (z \dot{-} base)) \\
&\quad \sqcup((y \dot{-} base) \sqcap (x \dot{-} base)) \sqcup ((y \dot{-} base) \sqcap x \sqcap base \sqcap z) \sqcup ((y \dot{-} base) \sqcap (z \dot{-} base) \sqcap x) \\
&\quad \sqcup((z \dot{-} base) \dot{-} x) \qquad (*)
\end{aligned}$$

Note that each term in (*) is dominated by a term of $(x \dot{-} base) \sqcup (y \dot{-} base) \sqcup (z \dot{-} base) \sqcup (x \sqcap y \sqcap z \sqcap base)$; thus, $(x\,[base\,]y)[x\,](x\,[base\,]z) \sqsubseteq (x\,[base\,]y,\, z)$.

However, by continuing from (*), we can also show that $(x\,[base\,]y)[x\,](x\,[base\,]z) \sqsupseteq (x\,[base\,]y,\, z)$.

$$\begin{aligned}
&= (x \dot{-} base) \\
&\quad \sqcup((y \dot{-} base) \dot{-} x) \sqcup ((y \dot{-} base) \sqcap (x \dot{-} base)) \sqcup ((y \dot{-} base) \sqcap x \sqcap base \sqcap z) \sqcup ((y \dot{-} base) \sqcap (z \dot{-} base) \sqcap x) \\
&\quad \sqcup((z \dot{-} base) \dot{-} x) \sqcup ((z \dot{-} base) \sqcap (x \dot{-} base)) \sqcup ((z \dot{-} base) \sqcap x \sqcap base \sqcap y) \sqcup ((z \dot{-} base) \sqcap (y \dot{-} base) \sqcap x) \\
&\quad \sqcup(x \sqcap base \sqcap z \sqcap y) \qquad \text{because } (x \dot{-} base) \sqcap x = x \dot{-} base \\
&= (x \dot{-} base) \\
&\quad \sqcup(((y \dot{-} base) \sqcup (z \dot{-} base)) \dot{-} x) \\
&\quad \sqcup(((y \dot{-} base) \sqcup (z \dot{-} base)) \sqcap (x \dot{-} base)) \\
&\quad \sqcup((x \sqcap base) \sqcap (((y \dot{-} base) \sqcap z) \sqcup ((z \dot{-} base) \sqcap y))) \\
&\quad \sqcup(((y \dot{-} base) \sqcap (z \dot{-} base)) \sqcap x) \\
&\quad \sqcup(x \sqcap base \sqcap z \sqcap y) \qquad (**)
\end{aligned}$$

Now consider the second and third terms of the expression in line (**). Their join is of the form $(a \dot{-} x) \sqcup (a \sqcap (x \dot{-} base))$, where $a = (y \dot{-} base) \sqcup (z \dot{-} base) = (y \sqcup z) \dot{-} base$.

$$\begin{aligned}
(a \dot{-} x) \sqcup (a \sqcap (x \dot{-} base)) &\sqsupseteq (a \dot{-} x) \sqcup ((a \sqcap x) \dot{-} base) && \text{by Proposition A.25} \\
&\sqsupseteq ((a \dot{-} x) \dot{-} base) \sqcup ((a \sqcap x) \dot{-} base) && \text{by Proposition A.11} \\
&= ((a \dot{-} x) \sqcup (a \sqcap x)) \dot{-} base && \text{by Proposition A.14} \\
&= a \dot{-} base && \text{by Proposition A.23} \\
&= ((y \sqcup z) \dot{-} base) \dot{-} base \\
&= (y \sqcup z) \dot{-} base \\
&= (y \dot{-} base) \sqcup (z \dot{-} base)
\end{aligned}$$

Substituting into (**), we have

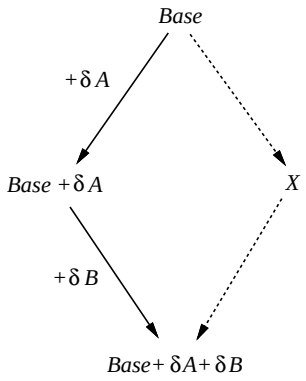
$$\begin{aligned}
(x\,[base\,]y)[x\,](x\,[base\,]z) &\sqsupseteq (x \dot{-} base) \sqcup (y \dot{-} base) \sqcup (z \dot{-} base) \sqcup (x \sqcap y \sqcap z \sqcap base) \\
&= (x\,[base\,]y,\, z)
\end{aligned}$$

We have shown that $(x\,[base\,]y,\, z) \sqsupseteq (x\,[base\,]y)[x\,](x\,[base\,]z) \sqsupseteq (x\,[base\,]y,\, z)$; consequently, $(x\,[base\,]y)[x\,](x\,[base\,]z) = (x\,[base\,]y,\, z)$. □

4.3. Compatible Integrands

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.

Example. Consider the case of two consecutive edits to base program *Base*; let $Base + \delta A$ be the result of modifying *Base*, and let $Base + \delta A + \delta B$ be the result of modifying $Base + \delta A$. By "separating consecutive edits," we mean creating a program $Base + \delta B$ that includes the second modification but not the first.



One way of formalizing this goal is to say that we are looking for an integrand X that is compatible with base *Base*, integrand $Base + \delta A$, and result $Base + \delta A + \delta B$; that is, X should satisfy the equation $(Base + \delta A)[Base](X) = Base + \delta A + \delta B$.

In this section the algebraic approach introduced in the previous sections is used to study this question. The question is posed as "When does there exist an integrand compatible with a given base *base*, integrand *a*, and result *m*?," or, equivalently, "When does there exist an element *x* such that $a\,[base\,]x = m$?" We show that, if they exist, the solutions to $a\,[base\,]x = m$ form a meet semi-lattice with a least element, and we give a closed formula for the least element. (These results are further extended in Section 5.4.)

Existence of a Compatible Integrand

The theorem proven in this section provides a test for the existence of a compatible integrand. It shows that a solution to the equation $a\,[base\,]x = m$ exists if and only if *m* itself is a suitable integrand (*i.e.*, if and only if *m* itself has the property that $a\,[base\,]m = m$).

Lemma 4.5. *$a\,[base\,]x = m$ iff $a \dot{-} (m \sqcup base) = \bot$ and $(m \dot{-} a) \dot{-} (m \dot{-} base) = \bot$.*

Proof.

$\Leftarrow$ case: Assuming $a \dot{-} (m \sqcup base) = \bot$ and $(m \dot{-} a) \dot{-} (m \dot{-} base) = \bot$, we must show that there is a solution to $a\,[base\,]x = m$.

We will show that there is a solution to $a\,[base\,]x = m$ by showing that *m* itself is a solution (*i.e.*, $a\,[base\,]m = m$). The proof breaks into two parts: in part (i), we show that $a\,[base\,]m \sqsubseteq m$; in part (ii), we show that $a\,[base\,]m \sqsupseteq m$.

(i) We will show that $a\,[base\,]m \sqsubseteq m$.
We start by considering the terms of $a\,[base\,]m$:
$a\,[base\,]m = (a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \dot{-} base)$.

(1) By the properties of $\sqcap$, we know $a \sqcap base \sqcap m \sqsubseteq m$.
(2) Because $a \dot{-} (m \sqcup base) = \bot$ we know $a \sqsubseteq m \sqcup base$, and consequently $a \dot{-} base \sqsubseteq m$.
(3) Because $m \sqsubseteq m \sqcup base$, we know $m \dot{-} base \sqsubseteq m$.

Thus, $a\,[base\,]m \sqsubseteq m \sqcup m \sqcup m = m$. (†)

(ii) We will show that $a\,[base\,]m \sqsupseteq m$.
From the assumption $(m \dot{-} a) \dot{-} (m \dot{-} base) = \bot$, we know that $(m \dot{-} a) \sqsubseteq (m \dot{-} base)$. (*)

$$\begin{aligned} m \div (a \sqcap base \sqcap m) &= (m \div a) \sqcup (m \div base) \sqcup (m \div m) && \text{by Proposition A.15} \\ &= (m \div a) \sqcup (m \div base) \sqcup \bot && \text{by Proposition A.4} \\ &= (m \div base) && \text{by (*)} \end{aligned}$$

We now start from $m \div base = m \div (a \sqcap base \sqcap m)$ and join both sides by $(a \sqcap base \sqcap m)$.

$$\begin{aligned} (m \div base) \sqcup (a \sqcap base \sqcap m) &= (m \div (a \sqcap base \sqcap m)) \sqcup (a \sqcap base \sqcap m) \\ &= m \sqcup (a \sqcap base \sqcap m) && \text{by Proposition A.12} \\ &= m && \text{(**)} \end{aligned}$$

$$\begin{aligned} a\,[base\,]m &= (a \div base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \div base) \\ &\sqsupseteq (a \sqcap base \sqcap m) \sqcup (m \div base) \\ &= m && \text{by (**)} \quad \text{(‡)} \end{aligned}$$

Combining (†) and (‡), we have $m \sqsupseteq a\,[base\,]m \sqsupseteq m$; hence, $a\,[base\,]m = m$.

$\Rightarrow$ *case: Assuming* $a\,[base\,]x = m$, *we must show that (i)* $a \div (m \sqcup base) = \bot$, *and (ii)* $(m \div a) \div (m \div base) = \bot$.

We consider each case in turn below.

(i) If $a\,[base\,]x = m$, then we have

$$\begin{aligned} \bot &= m \div m \\ &= a\,[base\,]x \div m \\ &= ((a \div base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \div base)) \div m \\ &= ((a \div base) \div m) \sqcup ((a \sqcap base \sqcap x) \div m) \sqcup ((x \div base) \div m) && \text{by Proposition A.14} \\ &\sqsupseteq (a \div base) \div m \\ &= a \div (m \sqcup base) && \text{by Proposition A.16} \end{aligned}$$

Therefore, $a \div (m \sqcup base) = \bot$.

(ii) If $a\,[base\,]x = m$, then

$$\begin{aligned} (m \div a) \div (m \div base) &= (a\,[base\,]x \div a) \div (a\,[base\,]x \div base) \\ &= [((a \div base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \div base)) \div a\,] \\ &\qquad \div [((a \div base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \div base)) \div base\,] \\ &= [((a \div base) \div a) \sqcup ((a \sqcap base \sqcap x) \div a) \sqcup ((x \div base) \div a)] \\ &\qquad \div [((a \div base) \div base) \sqcup ((a \sqcap base \sqcap x) \div base) \sqcup ((x \div base) \div base)] \\ & && \text{by Proposition A.14} \end{aligned}$$

It is possible to simplify five of the six terms in the last expression.

$(a \div base) \div a = (a \div a) \div base = \bot \div base = \bot$ by Propositions A.16, A.4, and A.5

$(a \sqcap base \sqcap x) \sqsubseteq a$, so $(a \sqcap base \sqcap x) \div a = \bot$ by Proposition A.2

$(a \div base) \div base = a \div (base \sqcup base) = a \div base$ by Proposition A.16

$(a \sqcap base \sqcap x) \sqsubseteq base$, so $(a \sqcap base \sqcap x) \div base = \bot$ by Proposition A.2

$(x \div base) \div base = x \div (base \sqcup base) = x \div base$ by Proposition A.16

Picking up the derivation, we have

$$\begin{aligned} (m \div a) \div (m \div base) &= [\bot \sqcup \bot \sqcup ((x \div base) \div a)] \div [(a \div base) \sqcup \bot \sqcup (x \div base)] \\ &= (x \div (a \sqcup base)) \div ((a \div base) \sqcup (x \div base)) \\ &= x \div ((a \sqcup base) \sqcup (a \div base) \sqcup (x \div base)) && \text{by Proposition A.16} \\ &= x \div (a \sqcup base \sqcup (x \div base)) && \text{by Proposition A.11} \\ &= ((x \div base) \div (x \div base)) \div a && \text{by Proposition A.16} \\ &= \bot \div a && \text{by Proposition A.4} \\ &= \bot && \text{by Proposition A.5} \end{aligned}$$

□

Theorem 4.6. *$a\,[base\,]x = m$ iff $a\,[base\,]m = m$.*

Proof.

$\Leftarrow$ *case:*

The proof of this case is immediate: To show that $a\,[base\,]x = m$ has a solution we merely choose x to be m.

$\Rightarrow$ *case:*

If $a\,[base\,]x = m$, then by the previous lemma, $a \dot{-} (m \sqcup base) = \bot$ and $(m \dot{-} a) \dot{-} (m \dot{-} base) = \bot$. As shown in the proof of the $\Leftarrow$ case of the lemma, if $a \dot{-} (m \sqcup base) = \bot$ and $(m \dot{-} a) \dot{-} (m \dot{-} base) = \bot$, then $a\,[base\,]m = m$.

Existence of a Minimum Compatible Integrand

In this section, we give a closed formula for the least solution of the equation $a\,[base\,]x = m$ and show that it is, in fact, the least solution.

Lemma 4.7. *If $a\,[base\,]m = m$ then $m \dot{-} a = m \dot{-} (a \sqcup base)$.*

Proof. Because $a\,[base\,]m = m$, we know from Theorem 4.6 and Lemma 4.5 that $a \dot{-} (m \sqcup base) = \bot$ and $(m \dot{-} a) \dot{-} (m \dot{-} base) = \bot$.

$$(m \dot{-} a) \dot{-} (m \dot{-} base) = \bot$$
$$m \dot{-} a \sqsubseteq m \dot{-} base$$
$$m \dot{-} a = (m \dot{-} a) \dot{-} a \sqsubseteq (m \dot{-} base) \dot{-} a = m \dot{-} (a \sqcup base) \qquad (*)$$

However, by Proposition A.11 we know that

$$m \dot{-} a \sqsupseteq (m \dot{-} a) \dot{-} base = m \dot{-} (a \sqcup base) \qquad (**)$$

Therefore, by (*) and (**), $m \dot{-} a = m \dot{-} (a \sqcup base)$. $\square$

Lemma 4.8. *If $a\,[base\,]m = m$ then $(a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \dot{-} a) = m$.*

Proof. The proof breaks into two parts: in part (i), we show that $(a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \dot{-} a) \sqsubseteq m$; in part (ii), we show that $(a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \dot{-} a) \sqsupseteq m$.

(i) Because $a\,[base\,]m = m$, we know that $a \dot{-} (m \sqcup base) = \bot$. Consequently, $a \sqsubseteq m \sqcup base$, or equivalently $a \dot{-} base \sqsubseteq m$. Thus, $(a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \dot{-} a) \sqsubseteq m \sqcup m \sqcup m = m$.

(ii) By Proposition A.18, $(a \dot{-} base) \sqcup (m \dot{-} a) \sqsupseteq m \dot{-} base$; thus,

$$(a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \dot{-} a) \sqsupseteq (a \sqcap base \sqcap m) \sqcup (m \dot{-} base).$$

From part (ii) of the $\Leftarrow$ case of Lemma 4.5, we know that from $(m \dot{-} a) \dot{-} (m \dot{-} base) = \bot$ we can deduce $(a \sqcap base \sqcap m) \sqcup (m \dot{-} base) = m$. Therefore, $(a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \dot{-} a) \sqsupseteq m$. $\square$

Definition 4.9. $x_{min} \triangleq (m \dot{-} a) \sqcup ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base))$.

Theorem 4.10. *If $a\,[base\,]m = m$ then x_{min} is the minimum x such that $a\,[base\,]x = m$.*

Proof.

Part I. Show that $a\,[base\,]x_{min} = m$.

$$a\,[base\,]x_{min} = (a \dot{-} base) \sqcup (a \sqcap base \sqcap ((m \dot{-} a) \sqcup ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base))))$$
$$\sqcup (((m \dot{-} a) \sqcup ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base))) \dot{-} base) \qquad (*)$$

First, we simplify the second term of (*).

$$
\begin{aligned}
&a \sqcap base \sqcap ((m \dot{-} a) \sqcup ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base))) \\
&\quad = (a \sqcap base \sqcap (m \dot{-} a)) \sqcup (a \sqcap base \sqcap ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base))) \\
&\quad = (a \sqcap base \sqcap (m \dot{-} a)) \sqcup ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base)) && \text{because } a \sqcap base \sqsupseteq a \sqcap base \sqcap m
\end{aligned}
$$

Next, we simplify the third term of (*).

$$
\begin{aligned}
&((m \dot{-} a) \sqcup ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base))) \dot{-} base \\
&\quad = ((m \dot{-} a) \dot{-} base) \sqcup (((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base)) \dot{-} base) \\
&\quad = (m \dot{-} (a \sqcup base)) \sqcup (((a \sqcap base \sqcap m) \dot{-} base) \dot{-} (a \dot{-} base)) \\
&\quad = (m \dot{-} (a \sqcup base)) \sqcup (\bot \dot{-} (a \dot{-} base)) \\
&\quad = (m \dot{-} (a \sqcup base)) \sqcup \bot \\
&\quad = m \dot{-} a && \text{by Lemma 4.7}
\end{aligned}
$$

$$
\begin{aligned}
a\,[base\,]x_{min} &= (a \dot{-} base) \sqcup [(a \sqcap base \sqcap (m \dot{-} a)) \sqcup ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base))] \sqcup (m \dot{-} a) \\
&= (a \dot{-} base) \sqcup ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base)) \sqcup (m \dot{-} a) \sqcup (a \sqcap base \sqcap (m \dot{-} a)) \\
&= (a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \dot{-} a) \sqcup (a \sqcap base \sqcap (m \dot{-} a)) && \text{by Proposition A.12}
\end{aligned}
$$

By Lemma 4.8, we know that $(a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqcup (m \dot{-} a) = m$. Thus we can continue the derivation above as follows:

$$
\begin{aligned}
&= m \sqcup (a \sqcap base \sqcap (m \dot{-} a)) \\
&= m
\end{aligned}
$$

This completes the proof of Part I; we have shown that $a\,[base\,]x_{min} = m$.

Part II. Show that x_{min} is the minimum x such that $a\,[base\,]x = m$.

Suppose that x is an element such that $a\,[base\,]x = m$. We will demonstrate that $x \sqsupseteq x_{min}$.

$$
\begin{aligned}
\bot &= a\,[base\,]x_{min} \dot{-} a\,[base\,]x \\
&= [(a \dot{-} base) \sqcup (a \sqcap base \sqcap x_{min}) \sqcup (x_{min} \dot{-} base)] \dot{-} [(a \dot{-} base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \dot{-} base)] \\
&= [(a \dot{-} base) \dot{-} ((a \dot{-} base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \dot{-} base))] \\
&\qquad \sqcup [(a \sqcap base \sqcap x_{min}) \dot{-} ((a \dot{-} base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \dot{-} base))] \\
&\qquad \sqcup [(x_{min} \dot{-} base) \dot{-} ((a \dot{-} base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \dot{-} base))] && (*)
\end{aligned}
$$

Note that, by Proposition A.2, the first term equals $\bot$. In addition, because the outermost connectives in (*) are joins, each of the remaining two terms must also equal $\bot$.

First, consider the third term of (*).

$$
\begin{aligned}
x_{min} \dot{-} base &\sqsubseteq (a \dot{-} base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \dot{-} base) \\
x_{min} &\sqsubseteq (a \dot{-} base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \dot{-} base) \sqcup base \\
&= (a \sqcup base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \dot{-} base) \\
&= a \sqcup base \sqcup x \sqcup (a \sqcap base \sqcap x) \\
&= a \sqcup base \sqcup x
\end{aligned}
$$

Therefore, $x_{min} \dot{-} (a \sqcup base) \sqsubseteq x$, or, expanding with the definition of x_{min},

$$((m \dot{-} a) \sqcup ((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base))) \dot{-} (a \sqcup base) \sqsubseteq x.$$

$$
\begin{aligned}
x &\sqsupseteq ((m \dot{-} a) \dot{-} (a \sqcup base)) \sqcup (((a \sqcap base \sqcap m) \dot{-} (a \dot{-} base)) \dot{-} (a \sqcup base)) \\
&= (((m \dot{-} a) \dot{-} a) \dot{-} base) \sqcup (((a \sqcap base \sqcap m) \dot{-} (a \sqcup base)) \dot{-} (a \dot{-} base)) \\
&= ((m \dot{-} a) \dot{-} base) \sqcup (\bot \dot{-} (a \dot{-} base)) \\
&= (m \dot{-} (a \sqcup base)) \sqcup \bot \\
&= m \dot{-} a && \text{by Lemma 4.7}
\end{aligned}
$$

Thus, $x \sqsupseteq m \dot{-} a$. (†)

Now consider the second term of (*).

$$a \sqcap base \sqcap x_{min} \sqsubseteq (a \div base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \div base)$$
$$(a \sqcap base \sqcap x_{min}) \div (a \div base) \sqsubseteq (a \sqcap base \sqcap x) \sqcup (x \div base) \sqsubseteq x$$
$$x \sqsupseteq (a \sqcap base \sqcap x_{min}) \div (a \div base)$$
$$= \{a \sqcap base \sqcap ((m \div a) \sqcup ((a \sqcap base \sqcap m) \div (a \div base)))\} \div (a \div base)$$
$$= \{(a \sqcap base \sqcap (m \div a)) \sqcup (a \sqcap base \sqcap ((a \sqcap base \sqcap m) \div (a \div base)))\} \div (a \div base)$$
$$= \{(a \sqcap base \sqcap (m \div a)) \sqcup ((a \sqcap base \sqcap m) \div (a \div base))\} \div (a \div base)$$
because $a \sqcap base \sqsupseteq a \sqcap base \sqcap m$
$$= ((a \sqcap base \sqcap (m \div a)) \div (a \div base)) \sqcup (((a \sqcap base \sqcap m) \div (a \div base)) \div (a \div base))$$
$$= ((a \sqcap base \sqcap (m \div a)) \div (a \div base)) \sqcup ((a \sqcap base \sqcap m) \div (a \div base))$$
$$\sqsupseteq (a \sqcap base \sqcap m) \div (a \div base) \qquad (‡)$$

By (†), $x \sqsupseteq m \div a$.
By (‡), $x \sqsupseteq (a \sqcap base \sqcap m) \div (a \div base)$.
Therefore, $x \sqsupseteq (m \div a) \sqcup ((a \sqcap base \sqcap m) \div (a \div base)) = x_{min}$.
□

Properties of Solutions of $a\,[base\,]x = m$

Lemma 4.11. *Solutions of $a\,[base\,]x = m$ are closed under $\sqcap$.*

Proof. Let x_1 and x_2 be two solutions of $a\,[base\,]x = m$ (*i.e.*, $a\,[base\,]x_1 = m$ and $a\,[base\,]x_2 = m$). The proof breaks into two parts: in part (i), we show that $a\,[base\,](x_1 \sqcap x_2) \sqsubseteq m$; in part (ii), we show that $a\,[base\,](x_1 \sqcap x_2) \sqsupseteq m$.

(i)

$$a\,[base\,](x_1 \sqcap x_2) \sqsubseteq a\,[base\,]x_1 \sqcap a\,[base\,]x_2$$
$$= m \sqcap m$$
$$= m \qquad (*)$$

(ii) Because $a\,[base\,]x_1 = m$ and $a\,[base\,]x_2 = m$, we know that $x_1 \sqsupseteq x_{min}$ and $x_2 \sqsupseteq x_{min}$; therefore, $x_1 \sqcap x_2 \sqsupseteq x_{min}$.

$$a\,[base\,](x_1 \sqcap x_2) = (a \div base) \sqcup (a \sqcap base \sqcap (x_1 \sqcap x_2)) \sqcup ((x_1 \sqcap x_2) \div base)$$
$$\sqsupseteq (a \div base) \sqcup (a \sqcap base \sqcap x_{min}) \sqcup (x_{min} \div base)$$
$$= m \qquad (**)$$

Combining (*) and (**), we have $m \sqsubseteq a\,[base\,](x_1 \sqcap x_2) \sqsubseteq m$; hence, $a\,[base\,](x_1 \sqcap x_2) = m$. □

Theorem 4.12. *Solutions of $a\,[base\,]x = m$ form a meet semi-lattice with least element x_{min}.*

Proof. Immediate from Lemma 4.11, together with Theorem 4.10. □

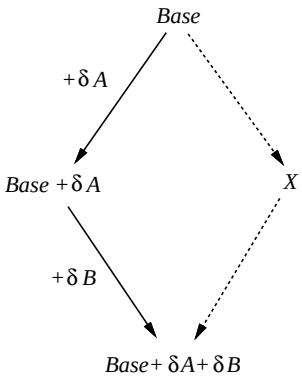
The question of when there is an integrand compatible with a given base *base*, integrand *a*, and result *m* is re-examined in Section 5.4, where the result just given as Theorem 4.12 is extended to Theorem 5.9.

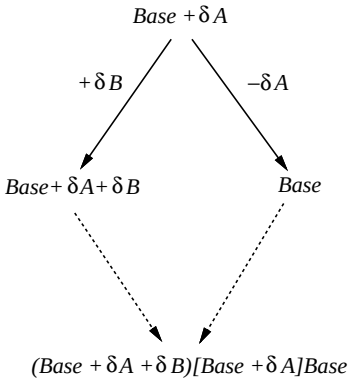
Separating Consecutive Edits by Re-Rooting

In this section, we consider a different approach to the problem of separating consecutive edits to a base program into individual edits on the base program.

Example. Consider again the case of two consecutive edits to a base program *Base*, where $Base + \delta A$ is the result of modifying *Base*, $Base + \delta A + \delta B$ is the result of modifying $Base + \delta A$, and we want to create a program $Base + \delta B$ that includes the second modification but not the first. We now consider an alternative approach to that of solving an equation (as shown below on the left and discussed earlier). This time our approach is to re-root the development history, as shown below on the right, so that $Base + \delta A$ rather than

Base is treated as the base program.





Programs *Base* and $Base + \delta A + \delta B$ are treated as two variants of $Base + \delta A$. For instance, instead of treating the differences between *Base* and $Base + \delta A$ as changes $+\delta A$ that were made to *Base* to create $Base + \delta A$, they are now treated as changes $-\delta A$ made to $Base + \delta A$ to create *Base*.[12] For instance, when *Base* is the base program, a statement *s* that occurs in $Base + \delta A$ but not in *Base* is a "new" statement arising from an insertion; when $Base + \delta A$ is the base program, we treat the missing *s* in *Base* as if a programmer had deleted *s* from $Base + \delta A$ to create *Base*. (The status of variant $Base + \delta A + \delta B$ is unchanged; it is still treated as a variant derived from $Base + \delta A$.) $Base + \delta B$ is created by integrating *Base* and $Base + \delta A + \delta B$ with respect to base program $Base + \delta A$ (*i.e.*, by performing the integration $Base\,[Base + \delta A\,](Base + \delta A + \delta B)$).

In this section, our algebraic techniques are used to demonstrate that the re-rooting approach is, in fact, reasonable. Below, we show that the result of integrating after re-rooting is not, in general, an integrand compatible with base *Base*, integrand $Base + \delta A$, and result $Base + \delta A + \delta B$ (even when a compatible integrand does exist); however, the element *E* produced by integrating after re-rooting (where $E = Base\,[Base + \delta A\,](Base + \delta A + \delta B)$) has the property that $(Base + \delta A)[Base\,]E \sqsupseteq Base + \delta A + \delta B$. This assures us that *E* captures everything that is different between $Base + \delta A + \delta B$ and $Base + \delta A$ (*i.e.*, all of the $+\delta B$ change, as desired), plus more. We then show that whenever a compatible integrand exists, *E* is greater than or equal to some compatible integrand; in particular, we show that taking the meet of *E* and $Base + \delta A + \delta B$ produces one of the compatible integrands.

Theorem 4.13. $a\,[base\,]m \sqsubseteq a\,[base\,](m\,[a\,]base)$.

Proof.

$$
\begin{aligned}
a\,[base\,](m\,[a\,]base) &= (a \div base) \sqcup (a \sqcap base \sqcap m\,[a\,]base) \sqcup (m\,[a\,]base \div base) \\
&= (a \div base) \sqcup (a \sqcap base \sqcap ((m \div a) \sqcup (a \sqcap base \sqcap m) \sqcup (base \div a))) \sqcup ((m \div a) \div base) \\
&= (a \div base) \sqcup (a \sqcap base \sqcap (m \div a)) \sqcup (a \sqcap base \sqcap a \sqcap base \sqcap m) \sqcup (a \sqcap base \sqcap (base \div a)) \\
&\qquad \sqcup ((m \div a) \div base) \\
&= (a \div base) \sqcup (a \sqcap base \sqcap m) \sqcup (a \sqcap (base \div a)) \sqcup ((m \div a) \div base) \\
&= ((a \sqcup (m \div a)) \div base) \sqcup (a \sqcap base \sqcap m) \sqcup (a \sqcap (base \div a)) \\
&= ((a \sqcup m) \div base) \sqcup (a \sqcap base \sqcap m) \sqcup (a \sqcap (base \div a)) \qquad \text{by Proposition A.12}
\end{aligned}
$$

[12]The notations $+\delta A$ and $-\delta A$ are used here informally to suggest editing operations that, respectively, add and remove a feature from a program. They are not intended as formal operators. The purpose of the discussion is merely to motivate the idea that re-rooting and then integrating is potentially useful. The formal characterization of the re-rooting approach is the subject of the rest of the section.

$$= (a \div base) \sqcup (m \div base) \sqcup (a \sqcap base \sqcap m) \sqcup (a \sqcap (base \div a)) \qquad \text{by Proposition A.14}$$
$$\sqsupseteq a\,[base\,]m$$

□

Corollary 4.14. *If* $a\,[base\,]m = m$, *then* $m \sqsubseteq a\,[base\,](m\,[a\,]base)$.

Proof. Immediate from Theorem 4.13. □

We can show by means of an example that the $\sqsubseteq$ in the above corollary is, at times, strict.

Example. The programs shown below have the properties that $a\,[base\,]m = m$ and $m \sqsubset a\,[base\,](m\,[a\,]base)$.

a	$base$	m	$a\,[base\,]m$	$m\,[a\,]base$	$a\,[base\,](m\,[a\,]base)$
$DC\lceil$**program** $x := 1;$ $y := x$ **end**()$\rfloor$	$DC\lceil$**program** $x := 1;$ $y := x;$ $z := y$ **end**()$\rfloor$	$DC\lceil$**program** $x := 1;$ $v := 2;$ $w := v$ **end**()$\rfloor$	$DC\lceil$**program** $x := 1;$ $v := 2;$ $w := v$ **end**()$\rfloor$	$DC\lceil$**program** $x := 1;$ $y := x;$ $z := y;$ $v := 2;$ $w := v$ **end**()$\rfloor$	$DC\lceil$**program** $x := 1;$ $y := x;$ $v := 2;$ $w := v$ **end**()$\rfloor$

Note that m is strictly less than $a\,[base\,](m\,[a\,]base)$ due to the presence in $a\,[base\,](m\,[a\,]base)$ of the slice

program
  $x := 1;$
  $y := x$
end().

(Even though this slice is not a member of m, it does occur in a, $base$, and $m\,[a\,]base$; thus, it is part of $a \sqcap base \sqcap m\,[a\,]base$ and hence occurs in $a\,[base\,](m\,[a\,]base)$.)

The above example shows that $m\,[a\,]base$—the result of integrating after re-rooting—is not necessarily a solution of $a\,[base\,]x = m$ (when a solution exists). We now show that $m\,[a\,]base$ is greater than or equal to some solution of $a\,[base\,]x = m$; in particular, we show that when there exists a solution of $a\,[base\,]x = m$, $(m\,[a\,]base) \sqcap m$ is one of the solutions.

Lemma 4.15. $(a\,[base\,]b) \sqcap a \sqsupseteq a\,[base\,](a \sqcap b)$.

Proof.

$$\begin{aligned}(a\,[base\,]b) \sqcap a &= ((a \div base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \div base)) \sqcap a \\ &= ((a \div base) \sqcap a) \sqcup (a \sqcap base \sqcap b) \sqcup ((b \div base) \sqcap a) \\ &= (a \div base) \sqcup (a \sqcap base \sqcap b) \sqcup (a \sqcap (b \div base)) && \text{by Proposition A.11} \\ &\sqsupseteq (a \div base) \sqcup (a \sqcap base \sqcap b) \sqcup ((a \sqcap b) \div base) && \text{by Proposition A.25} \\ &= a\,[base\,](a \sqcap b)\end{aligned}$$

□

Theorem 4.16. $a\,[base\,]((m\,[a\,]base) \sqcap m) = a\,[base\,]m$.

Proof. The proof breaks into two parts: in part (i), we show that $a\,[base\,]((m\,[a\,]base) \sqcap m) \sqsubseteq a\,[base\,]m$; in part (ii), we show that $a\,[base\,]((m\,[a\,]base) \sqcap m) \sqsupseteq a\,[base\,]m$.

(i)

$$\begin{aligned} a\,[base\,]((m\,[a\,]base)\sqcap m) &\sqsubseteq a\,[base\,](m\,[a\,]base)\sqcap a\,[base\,]m && \text{by Proposition B.10}\\ &= a\,[base\,]m && (*)\\ & && \text{by Theorem 4.13}\end{aligned}$$

(ii)

$$\begin{aligned} a\,[base\,]((m\,[a\,]base)\sqcap m) &= (a \dot{-} base)\sqcup(a\sqcap base\sqcap((m\,[a\,]base)\sqcap m)\sqcap m)\sqcup(((m\,[a\,]base)\sqcap m)\dot{-} base)\\ &= (a \dot{-} base)\sqcup(a\sqcap base\sqcap m)\sqcup(((m\,[a\,]base)\sqcap m)\dot{-} base)\\ &\sqsupseteq (a \dot{-} base)\sqcup(a\sqcap base\sqcap m)\sqcup(m\,[a\,](base\sqcap m)\dot{-} base)\\ & && \text{by Lemma 4.15}\\ &= (a \dot{-} base)\sqcup(a\sqcap base\sqcap m)\sqcup((m \dot{-} a)\dot{-} base)\\ &\quad \sqcup((m\sqcap a\sqcap m\sqcap base)\dot{-} base)\sqcup(((base\sqcap m)\dot{-} a)\dot{-} base)\\ &= (a \dot{-} base)\sqcup(a\sqcap base\sqcap m)\sqcup((m \dot{-} a)\dot{-} base)\sqcup\bot\sqcup\bot\\ &= ((a\sqcup(m \dot{-} a))\dot{-} base)\sqcup(a\sqcap base\sqcap m)\\ &= ((a\sqcup m)\dot{-} base)\sqcup(a\sqcap base\sqcap m) && \text{by Proposition A.13}\\ &= (a \dot{-} base)\sqcup(m \dot{-} base)\sqcup(a\sqcap base\sqcap m)\\ &= a\,[base\,]m && (**)\end{aligned}$$

Combining (*) and (**), we have $a\,[base\,]m \sqsubseteq a\,[base\,]((m\,[a\,]base)\sqcap m)\sqsubseteq a\,[base\,]m$; hence, $a\,[base\,]((m\,[a\,]base)\sqcap m) = a\,[base\,]m$. □

Corollary 4.17. *If* $a\,[base\,]m = m$ *then* $(m\,[a\,]base)\sqcap m$ *is a solution of* $a\,[base\,]x = m$.

Proof. Immediate from Theorem 4.16. □

4.4. A Compatible Base

We now turn to the question of whether there is a base element compatible with given integrands a and b and result element m; that is, we want to know "When does there exist an element x such that $a\,[x\,]b = m$?"

Existence of a Compatible Base

Note that, because $a\,[x\,]b$ is anti-monotonic in x, we have

$a\sqcap b = a\,[\top\,]b \sqsubseteq a\,[x\,]b = m = a\,[x\,]b \sqsubseteq a\,[\bot\,]b = a\sqcup b$.

Lemma 4.18. $a\,[x\,]b = m$ *has a solution for* x *iff* $(a \dot{-} (a \dot{-} m))\dot{-}(a\sqcap b) = m \dot{-} b$ *and* $(b \dot{-} (b \dot{-} m))\dot{-}(b\sqcap a) = m \dot{-} a$.

Proof.

$\Rightarrow$ *case:*

Assuming that $a\,[x\,]b = m$ has a solution for x, we must show that $(a \dot{-} (a \dot{-} m))\dot{-}(a\sqcap b) = m \dot{-} b$ and $(b \dot{-} (b \dot{-} m))\dot{-}(b\sqcap a) = m \dot{-} a$. The proof breaks into two parts: in part (i), we show that $(a \dot{-} (a \dot{-} m))\dot{-}(a\sqcap b)\sqsubseteq m \dot{-} b$; in part (ii), we show that $(a \dot{-} (a \dot{-} m))\dot{-}(a\sqcap b)\sqsupseteq m \dot{-} b$. (The identical arguments with a and b interchanged show that $(b \dot{-} (b \dot{-} m))\dot{-}(b\sqcap a) = m \dot{-} a$.)

(i)

$$\begin{aligned} (a \dot{-} (a \dot{-} m))\dot{-}(a\sqcap b) &= ((a \dot{-} (a \dot{-} m))\dot{-} a)\sqcup((a \dot{-} (a \dot{-} m))\dot{-} b)\\ &= \bot\sqcup((a \dot{-} b)\dot{-}(a \dot{-} m))\\ &= (a \dot{-} b)\dot{-}(a \dot{-} m)\\ &\sqsubseteq m \dot{-} b && \text{by Proposition A.19}\end{aligned}$$

(ii)

$$
\begin{aligned}
(a \dot{-} (a \dot{-} m)) \dot{-} (a \sqcap b) &= ((a \dot{-} (a \dot{-} m)) \dot{-} a) \sqcup ((a \dot{-} (a \dot{-} m)) \dot{-} b) \\
&= \bot \sqcup ((a \dot{-} b) \dot{-} (a \dot{-} m)) \\
&= (a \dot{-} b) \dot{-} (a \dot{-} m)
\end{aligned}
$$

$$
\begin{aligned}
(m \dot{-} b) \dot{-} ((a \dot{-} (a \dot{-} m)) \dot{-} (a \sqcap b)) &= (m \dot{-} b) \dot{-} ((a \dot{-} b) \dot{-} (a \dot{-} m)) \\
&= (a\,[x\,]b \dot{-} b) \dot{-} ((a \dot{-} b) \dot{-} (a \dot{-} m)) \\
&= ((a \dot{-} x) \dot{-} b) \dot{-} ((a \dot{-} b) \dot{-} (a \dot{-} m)) \\
&= ((a \dot{-} b) \dot{-} x) \dot{-} ((a \dot{-} b) \dot{-} (a \dot{-} m)) \\
&\sqsubseteq (a \dot{-} m) \dot{-} x && \text{by Proposition A.19} \\
&= (a \dot{-} x) \dot{-} m \\
&\sqsubseteq ((a \dot{-} x) \dot{-} m) \sqcup ((a \sqcap x \sqcap b) \dot{-} m) \sqcup ((b \dot{-} x) \dot{-} m) \\
&= ((a \dot{-} x) \sqcup (a \sqcap x \sqcap b) \sqcup (b \dot{-} x)) \dot{-} m \\
&= a\,[x\,]b \dot{-} m \\
&= m \dot{-} m \\
&= \bot
\end{aligned}
$$

Therefore, $(a \dot{-} (a \dot{-} m)) \dot{-} (a \sqcap b) \sqsupseteq m \dot{-} b$. (**)

Putting (*) and (**) together, we have $m \dot{-} b \sqsupseteq (a \dot{-} (a \dot{-} m)) \dot{-} (a \sqcap b) \sqsupseteq m \dot{-} b$, hence $(a \dot{-} (a \dot{-} m)) \dot{-} (a \sqcap b) = m \dot{-} b$.

$\Leftarrow$ *case:*

Assuming that $(a \dot{-} (a \dot{-} m)) \dot{-} (a \sqcap b) = m \dot{-} b$ and $(b \dot{-} (b \dot{-} m)) \dot{-} (b \sqcap a) = m \dot{-} a$, we must show that $a\,[x\,]b = m$ has a solution for x. We will show that a solution exists by showing that $(a \dot{-} m) \sqcup (b \dot{-} m)$ is a solution.

The proof breaks into two parts: in part (i), we show that $a\,[(a \dot{-} m) \sqcup (b \dot{-} m)]b \sqsubseteq m$; in part (ii), we show that $a\,[(a \dot{-} m) \sqcup (b \dot{-} m)]b \sqsupseteq m$.

(i)

$$
\begin{aligned}
a\,[(a \dot{-} m) \sqcup (b \dot{-} m)]b &= (a \dot{-} ((a \dot{-} m) \sqcup (b \dot{-} m))) \sqcup (a \sqcap ((a \dot{-} m) \sqcup (b \dot{-} m)) \sqcap b) \\
&\qquad \sqcup (b \dot{-} ((a \dot{-} m) \sqcup (b \dot{-} m))) \\
&= ((a \dot{-} (a \dot{-} m)) \dot{-} (b \dot{-} m)) \sqcup (a \sqcap b \sqcap (a \dot{-} m)) \\
&\qquad \sqcup (a \sqcap b \sqcap (b \dot{-} m)) \sqcup ((b \dot{-} (b \dot{-} m)) \dot{-} (a \dot{-} m)) \\
&\sqsubseteq ((a \sqcap m) \dot{-} (b \dot{-} m)) \sqcup (a \sqcap (b \dot{-} m)) && \text{by Proposition A.27} \\
&\qquad \sqcup ((b \sqcap m) \dot{-} (a \dot{-} m)) \sqcup (b \sqcap (a \dot{-} m)) \\
&\sqsubseteq ((a \dot{-} (b \dot{-} m)) \sqcap (m \dot{-} (b \dot{-} m))) \sqcup (a \sqcap (b \dot{-} m)) && \text{by Proposition A.26} \\
&\qquad \sqcup ((b \dot{-} (a \dot{-} m)) \sqcap (m \dot{-} (a \dot{-} m))) \sqcup (b \sqcap (a \dot{-} m)) \\
&= (((a \dot{-} (b \dot{-} m)) \sqcup (a \sqcap (b \dot{-} m))) \sqcap ((m \dot{-} (b \dot{-} m)) \sqcup (a \sqcap (b \dot{-} m)))) \\
&\qquad \sqcup (((b \dot{-} (a \dot{-} m)) \sqcup (b \sqcap (a \dot{-} m))) \sqcap ((m \dot{-} (a \dot{-} m)) \sqcup (b \sqcap (a \dot{-} m)))) \\
&= (a \sqcap ((m \dot{-} (b \dot{-} m)) \sqcup (a \sqcap (b \dot{-} m)))) && \text{by Proposition A.23} \\
&\qquad \sqcup (b \sqcap ((m \dot{-} (a \dot{-} m)) \sqcup (b \sqcap (a \dot{-} m)))) \\
&= ((a \sqcap (m \dot{-} (b \dot{-} m))) \sqcup (a \sqcap (b \dot{-} m))) \\
&\qquad \sqcup ((b \sqcap (m \dot{-} (a \dot{-} m))) \sqcup (b \sqcap (a \dot{-} m))) \\
&= (a \sqcap ((m \dot{-} (b \dot{-} m)) \sqcup (b \dot{-} m))) \sqcup (b \sqcap ((m \dot{-} (a \dot{-} m)) \sqcup (a \dot{-} m))) \\
&= (a \sqcap (m \sqcup (b \dot{-} m))) \sqcup (b \sqcap (m \sqcup (a \dot{-} m))) && \text{by Proposition A.12} \\
&= (a \sqcap (m \sqcup b)) \sqcup (b \sqcap (m \sqcup a)) && \text{by Proposition A.12} \\
&= (a \sqcup b) \sqcap (a \sqcup (m \sqcup a)) \sqcap (b \sqcup (m \sqcup b)) \sqcap (m \sqcup a \sqcup b) \\
&= (a \sqcup b) \sqcap (a \sqcup m) \sqcap (b \sqcup m) && \text{because } m \sqsubseteq a \sqcup b \\
&= (a \sqcup b) \sqcap ((a \sqcap b) \sqcup (m \sqcap b) \sqcup (a \sqcap m) \sqcup m) \\
&= (a \sqcup b) \sqcap m && \text{because } a \sqcap b \sqsubseteq m \\
&= m && \text{because } m \sqsubseteq a \sqcup b
\end{aligned}
$$

Therefore, $m \sqsupseteq a\,[(a \dot{-} m) \sqcup (b \dot{-} m)]b$. (*)

(ii)

$$
\begin{aligned}
a\,[(a \dot{-} m) \sqcup (b \dot{-} m)]\,b &\sqsupseteq a\,[a\,[m\,]\,b\,]\,b && \text{by Proposition B.12}\\
&= a\,[(a \dot{-} m) \sqcup (a \sqcap m \sqcap b) \sqcup (b \dot{-} m)]\,b\\
&= a\,[(a \dot{-} m) \sqcup (a \sqcap b) \sqcup (b \dot{-} m)]\,b && \text{because } a \sqcap b \sqsubseteq m\\
&= (a \dot{-} ((a \dot{-} m) \sqcup (a \sqcap b) \sqcup (b \dot{-} m))) \sqcup (a \sqcap ((a \dot{-} m) \sqcup (a \sqcap b) \sqcup (b \dot{-} m)) \sqcap b)\\
&\qquad \sqcup (b \dot{-} ((a \dot{-} m) \sqcup (a \sqcap b) \sqcup (b \dot{-} m)))\\
&= (((a \dot{-} (a \dot{-} m)) \dot{-} (a \sqcap b)) \dot{-} (b \dot{-} m)) \sqcup (a \sqcap b)\\
&\qquad \sqcup (((b \dot{-} (b \dot{-} m)) \dot{-} (a \sqcap b)) \dot{-} (a \dot{-} m))\\
&= ((m \dot{-} b) \dot{-} (b \dot{-} m)) \sqcup (a \sqcap b) \sqcup ((m \dot{-} a) \dot{-} (a \dot{-} m)) && \text{by the assumptions}\\
&= (m \dot{-} b) \sqcup (a \sqcap b) \sqcup (m \dot{-} a) && \text{by Proposition A.30}\\
&= (m \dot{-} (a \sqcap b)) \sqcup (a \sqcap b) && \text{by Proposition A.15}\\
&= m
\end{aligned}
$$

Therefore, $a\,[(a \dot{-} m) \sqcup (b \dot{-} m)]\,b \sqsupseteq m$. (**)

Putting (*) and (**) together, we have $m \sqsupseteq a\,[(a \dot{-} m) \sqcup (b \dot{-} m)]\,b \sqsupseteq m$, hence, $a\,[(a \dot{-} m) \sqcup (b \dot{-} m)]\,b = m$. □

Existence of a Minimum Compatible Base

In this section, we show that when the equation $a\,[x\,]\,b = m$ has a solution for x, $(a \dot{-} m) \sqcup (b \dot{-} m)$ is the least solution.

Theorem 4.19. *If $a\,[x\,]\,b = m$ has a solution, then $(a \dot{-} m) \sqcup (b \dot{-} m)$ is the minimum x such that $a\,[x\,]\,b = m$.*

Proof. By the proof of Lemma 4.18, if the equation $a\,[x\,]\,b = m$ has a solution for x, then $(a \dot{-} m) \sqcup (b \dot{-} m)$ is a solution to the equation. It remains to be shown that $(a \dot{-} m) \sqcup (b \dot{-} m)$ is the *least* solution.

In the following derivation, let x be any solution to the equation $a\,[x\,]\,b = m$.

$$
\begin{aligned}
\bot &= m \dot{-} m\\
&= a\,[x\,]\,b \dot{-} m\\
&= ((a \dot{-} x) \sqcup (a \sqcap x \sqcap b) \sqcup (b \dot{-} x)) \dot{-} m\\
&= ((a \dot{-} x) \dot{-} m) \sqcup ((a \sqcap x \sqcap b) \dot{-} m) \sqcup ((b \dot{-} x) \dot{-} m)\\
&= ((a \dot{-} m) \dot{-} x) \sqcup ((a \sqcap x \sqcap b) \dot{-} m) \sqcup ((b \dot{-} m) \dot{-} x)\\
&\sqsupseteq ((a \dot{-} m) \dot{-} x) \sqcup ((b \dot{-} m) \dot{-} x)\\
&= ((a \dot{-} m) \sqcup (b \dot{-} m)) \dot{-} x
\end{aligned}
$$

Therefore, $((a \dot{-} m) \sqcup (b \dot{-} m)) \dot{-} x = \bot$, and hence $(a \dot{-} m) \sqcup (b \dot{-} m) \sqsubseteq x$. □

Properties of Solutions of $a\,[x\,]\,b = m$

Lemma 4.20. *Solutions of $a\,[x\,]\,b = m$ are closed under $\sqcap$.*

Proof. Let x_1 and x_2 be two solutions of $a\,[x\,]\,b = m$ (*i.e.*, $a\,[x_1\,]\,b = m$ and $a\,[x_2\,]\,b = m$).

$$
\begin{aligned}
a\,[x_1 \sqcap x_2\,]\,b &= a\,[x_1\,]\,b \sqcup a\,[x_2\,]\,b && \text{by (B.14)}\\
&= m \sqcup m\\
&= m
\end{aligned}
$$

□

Theorem 4.21. *Solutions of $a\,[x\,]\,b = m$ form a meet semi-lattice with least element $(a \dot{-} m) \sqcup (b \dot{-} m)$.*

Proof. Immediate from Lemma 4.20, together with Theorem 4.19. □

5. Algebraic Properties of Double Brouwerian Algebras

This section concerns *double Brouwerian algebras*—Brouwerian algebras whose duals are also Brouwerian algebras. Section 5.1 introduces the *quotient* operation, which is the dual of the pseudo-difference operation. In Section 5.2, it is shown that the lattice of downwards-closed sets of dependence-graph slices is a double Brouwerian algebra. Section 5.3 concerns the operation that is the dual of the

integration operation, and shows how the two operations are related. Section 5.4 re-examines the question of when there is an integrand compatible with a given base *base*, integrand *a*, and result *m* and generalizes the result that was given earlier in Section 4.3.

5.1. The Quotient Operation

Definition 5.1. For all $x, y \in DCS$, the *quotient* of x with respect to y, denoted by $x \div y$, is defined as

$$x \div y \triangleq \{ z \in G_1 \mid \nexists p \in (y - x) \text{ such that } p \leq z \}.$$

That is, $x \div y$ is the complement of the upwards closure of $y - x$.

In the definition of $x \div y$, elements of the set $y - x$ represent forbidden (sub-)slices of members of $x \div y$. The quotient operation is illustrated in Figure 11.

Example. The following table illustrates the differences between ordinary set difference ($-$), pseudo-difference ($\dot{-}$), and quotient ($\div$):

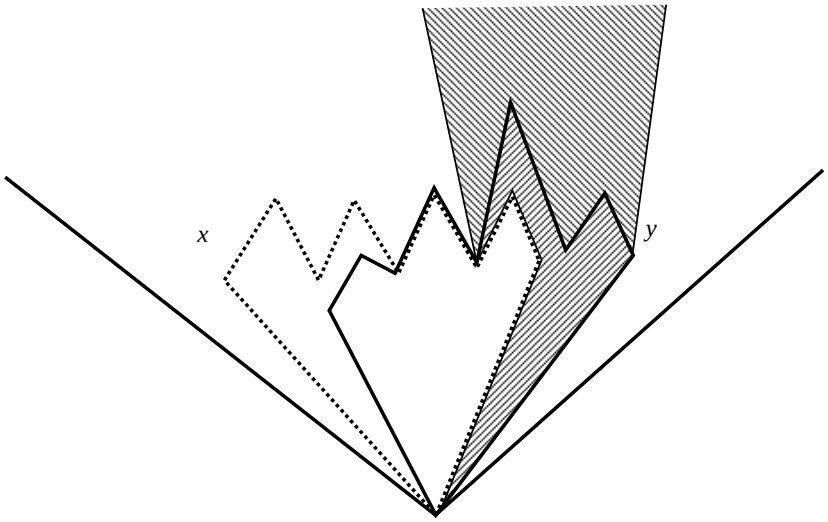


Figure 11. The value of $x \div y$ is the complement of the upwards closure of $y - x$; that is, $x \div y$ consists of all elements that do *not* dominate an element of $y - x$. Thus, $x \div y$ excludes both of the shaded regions. In the algebra of downwards-closed sets of single-point slices, elements of the set $y - x$ represent forbidden (sub-)slices of members of $x \div y$.

a	b	$a - b$	$b - a$
{**program** , **program** **end**() $x := 0$ **end**()}	{**program** , **program** , **program** **end**() $x := 0$ $x := 0;$ **end**() $y := x$ **end**()}	$\varnothing$ $(= \bot)$	{**program** $x := 0;$ $y := x$ **end**()}

$a \div b$	$b \div a$
$\varnothing$ $(= \bot)$	{**program** , **program** , **program** **end**() $x := 0$ $x := 0;$ **end**() $y := x$ **end**()}

$b \div a$	$a \div b$
G_1 $(= \top)$	$G_1 -$ {$s \in G_1$ | **program** $\leq s$ $x := 0;$ $y := x$ **end**()}

Note that because $b - a$ is the singleton set

{**program**
$x := 0;$
$y := x$
end()}

$a \div b$ is the (infinite) set of all single-point slices that do not contain the (sub-)slice

program
  $x := 0;$
  $y := x$
end().

5.2. Double Brouwerian Algebras

Definition 5.2. A *double Brouwerian algebra* [16] is an algebra $(L, \sqcup, \sqcap, \dot{-}, \div, \top)$ where both $(L, \sqcup, \sqcap, \dot{-}, \top)$ and $(L, \sqcap, \sqcup, \div, \top \dot{-} \top)$ are Brouwerian algebras. In particular,

(i) L is closed under $\div$.

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

Theorem 5.3. *(DCS, $\cup$, $\cap$, $\dot{-}$, $\div$, G_1) is a double Brouwerian algebra, where $\cup$ is set union and $\cap$ is set intersection.*

Proof. We must show that (DCS, $\cap$, $\cup$, $\div$, $\varnothing$) is a Brouwerian algebra, which involves showing (1) DCS is closed under $\div$ and (2) for all $a, b, c \in DCS$, $a \div b \supseteq c$ iff $a \supseteq b \cap c$.

To show property (1), consider any two elements $a, b \in DCS$. From Definition 5.1 (the definition of $\div$ in the algebra of downwards-closed sets of single-point slices) we see that, for all $q \in a \div b$, if z is a single-point slice of q, z is also a member of $a \div b$, hence $a \div b \in DCS$.

Two show property (2), there are two cases to consider.

$\Rightarrow$ *case: Assuming $a \div b \supseteq c$, we must show that $a \supseteq b \cap c$.*

Let $\overline{b}$ be the complement of b with respect to G_1 (*i.e.*, $\overline{b} = G_1 - b$). From Definition 5.1, we know that $\overline{b} \cup a \supseteq a \div b$.

$$\begin{aligned}
\bar{b} \cup a &\supseteq a \div b \supseteq c \\
(\bar{b} \cup a) \cap b &\supseteq b \cap c \\
(\bar{b} \cap b) \cup (a \cap b) &\supseteq b \cap c \\
\varnothing \cup (a \cap b) &\supseteq b \cap c \\
a \cap b &\supseteq b \cap c \\
a &\supseteq b \cap c.
\end{aligned}$$

$\Leftarrow$ *case: Assuming $a \supseteq b \cap c$, we must show that $a \div b \supseteq c$.*

Let z be a member of c; we will show that $z \in a \div b$. There are two cases to consider:

(1) Suppose $z \in b$. Because $z \in c$, $z \in b$, and $a \supseteq b \cap c$, we know $z \in a$. By the downwards-closure property of elements of *DCS*, $\forall p \in G_1$ such that $p \leq z$, $p \in a$. This means that $p \notin (b - a)$. Hence, $\nexists p \in (b - a)$ such that $p \leq z$; consequently, $z \in a \div b$.

(2) Suppose $z \notin b$. We first observe that $z \notin (b - a)$. Now suppose there exists a $p \in (b - a)$ such that $p \leq z$ (*). By the downwards-closure property of elements of *DCS*, because $p \leq z$ and $z \in c$, we know that $p \in c$. But since $p \in (b - a)$, we also have $p \in b$. Therefore $p \in b \cap c$, which means that $p \in a$ (because $a \supseteq b \cap c$). From $p \in a$ and $p \in b$, we conclude that $p \notin (b - a)$, which contradicts (*). Hence, $\nexists p \in (b - a)$ such that $p \leq z$; consequently, $z \in a \div b$.

□

5.3. An Alternative Way to Perform Integration

This section concerns a new criterion for program integration, based on the operation that is the dual of the integration operation. After introducing a few new concepts that are needed to define the dual operation, we show how the two operations are related.

Because the dependence-graph algebra is a *double* Brouwerian algebra, it is possible to perform integration using the dual of the operation $a\,[base\,]b$.

Definition 5.4. The *dual integration* of elements a and b with respect to element *base* is the element $a\,\{\,base\,\}b$ defined by $a\,\{\,base\,\}b \triangleq (a \div base) \sqcap (a \sqcup base \sqcup b) \sqcap (b \div base)$.

Note that, in the algebra of downwards-closed sets of single-point slices, if a, *base*, and b are all finite sets, then $a \sqcup base \sqcup b$ is finite. Hence, even though $a \div base$ and $b \div base$ are infinite sets, $a\,\{\,base\,\}b$ is guaranteed to be finite.

We now investigate how $a\,[base\,]b$ and $a\,\{\,base\,\}b$ are related; the theorem proven below shows that $a\,\{\,base\,\}b$ is always less than or equal to $a\,[base\,]b$. We then give an example for which strict inequality holds.

Theorem 5.5. $a\,\{\,base\,\}b \sqsubseteq a\,[base\,]b$.

Proof.

$$\begin{aligned}
a\,[base\,]b &= (a \dot{-} base) \sqcup (a \sqcap b) \sqcup (b \dot{-} base) && \text{by Corollary 4.2} \\
&= ((a \sqcup b) \dot{-} base) \sqcup (a \sqcap b) && \text{by Proposition A.14} \\
&= (a \sqcup ((a \sqcup b) \dot{-} base)) \sqcap (b \sqcup ((a \sqcup b) \dot{-} base)) \\
&= (a \sqcup (a \dot{-} base) \sqcup (b \dot{-} base)) \sqcap (b \sqcup (a \dot{-} base) \sqcup (b \dot{-} base)) && \text{by Proposition A.14} \\
&= (a \sqcup (b \dot{-} base)) \sqcap (b \sqcup (a \dot{-} base)) && \text{by Proposition A.11}
\end{aligned}$$

By the dual derivation, we have $a\,\{\,base\,\}b = (a \sqcap (b \div base)) \sqcup (b \sqcap (a \div base))$.

$$\begin{aligned}
a\,\{\,base\,\}b \dot{-} a\,[base\,]b &= [(a \sqcap (b \div base)) \sqcup (b \sqcap (a \div base))] \dot{-} [(a \sqcup (b \dot{-} base)) \sqcap (b \sqcup (a \dot{-} base))] \\
&= ((a \sqcap (b \div base)) \dot{-} [(a \sqcup (b \dot{-} base)) \sqcap (b \sqcup (a \dot{-} base))]) \\
&\quad \sqcup ((b \sqcap (a \div base)) \dot{-} [(a \sqcup (b \dot{-} base)) \sqcap (b \sqcup (a \dot{-} base))]) \\
&\quad && \text{by Proposition A.14} \\
&= ((a \sqcap (b \div base)) \dot{-} (a \sqcup (b \dot{-} base))) \sqcup ((a \sqcap (b \div base)) \dot{-} (b \sqcup (a \dot{-} base)))
\end{aligned}$$

$$\sqcup((b \sqcap (a \div base)) \dot{-} (a \sqcup (b \dot{-} base))) \sqcup ((b \sqcap (a \div base)) \dot{-} (b \sqcup (a \dot{-} base)))$$
by Proposition A.15
$$= ((a \sqcap (b \div base)) \dot{-} (b \sqcup (a \dot{-} base))) \sqcup ((b \sqcap (a \div base)) \dot{-} (a \sqcup (b \dot{-} base)))$$
by Proposition A.2

$$\begin{aligned}
(a \sqcap (b \div base)) \dot{-} (b \sqcup (a \dot{-} base)) &= ((a \sqcap (b \div base)) \dot{-} (a \dot{-} base)) \dot{-} b && \text{by Proposition A.16} \\
&\sqsubseteq ((b \div base) \sqcap (a \dot{-} (a \dot{-} base))) \dot{-} b && \text{by Proposition A.25} \\
&\sqsubseteq ((b \div base) \sqcap (a \sqcap base)) \dot{-} b && \text{by Proposition A.27} \\
&= (((b \div base) \sqcap base) \sqcap a) \dot{-} b \\
&= (b \sqcap base \sqcap a) \dot{-} b && \text{by the dual of Proposition A.12} \\
&= \bot && \text{by Proposition A.2}
\end{aligned}$$

Similarly, $(b \sqcap (a \div base)) \dot{-} (a \sqcup (b \dot{-} base)) = \bot$. Hence, $a\{\, base \,\}b \dot{-} a\,[base\,]b = \bot$, and thus, by Proposition A.2, $a\{\, base \,\}b \sqsubseteq a\,[base\,]b$. □

Example. Returning once again to the integration example from Sections 2.2 and 3.4, we can show that the inequality in the above theorem is, at times, strict; the sets of programs shown in the table in Figure 12 have the property that $DC(A)\{\, DC(Base) \,\}DC(B)$ is strictly less than $DC(A)[DC(Base)]DC(B)$. Because $DC(A)\{\, DC(Base) \,\}DC(B)$ is the *intersection* of $DC(A) \div DC(Base)$ with $(DC(A) \cup DC(Base) \cup DC(B)) \cap (DC(B) \div DC(Base))$, none of the following three single-point slices, which are all members of $DC(A)[DC(Base)]DC(B)$, occur in $DC(A)\{\, DC(Base) \,\}DC(B)$:

$DC(A)$	$DC(Base)$	$DC(B)$	
$DC\left\lceil \begin{array}{l} \textbf{program} \\ x := 0 \\ \textbf{end}(x) \end{array} \right\rfloor$	$DC\left\lceil \begin{array}{l} \textbf{program} \\ x := 0; \\ y := x \\ \textbf{end}(x, y) \end{array} \right\rfloor$	$DC\left\lceil \begin{array}{l} \textbf{program} \\ x := 0; \\ y := x; \\ z := y \\ \textbf{end}(x, y, z) \end{array} \right\rfloor$	
$DC(A) \dot{-} DC(Base)$	$DC(A) \cap DC(Base) \cap DC(B)$	$DC(B) \dot{-} DC(Base)$	$DC(A)[DC(Base)]DC(B)$
$\varnothing\ (= \bot)$	$DC\left\lceil \begin{array}{l} \textbf{program} \\ x := 0 \\ \textbf{end}(x) \end{array} \right\rfloor$	$DC\left\lceil \begin{array}{l} \textbf{program} \\ x := 0; \\ y := x; \\ z := y \\ \textbf{end}(z) \end{array} \right\rfloor$	$DC\left\lceil \begin{array}{l} \textbf{program} \\ x := 0; \\ y := x; \\ z := y \\ \textbf{end}(x, z) \end{array} \right\rfloor$
$DC(A) \div DC(Base)$	$DC(A) \cup DC(Base) \cup DC(B)$	$DC(B) \div DC(Base)$	$DC(A)\{\, DC(Base) \,\}DC(B)$
$G_1 - \left\{ s \in G_1 \mid \begin{array}{l} \textbf{program} \le s \\ x := 0; \\ y := x \\ \textbf{end}() \end{array} \right\}$	$DC\left\lceil \begin{array}{l} \textbf{program} \\ x := 0; \\ y := x; \\ z := y \\ \textbf{end}(x, y, z) \end{array} \right\rfloor$	$G_1\ (= \top)$	$DC\left\lceil \begin{array}{l} \textbf{program} \\ x := 0 \\ \textbf{end}(x) \end{array} \right\rfloor$

Figure 12. The table given above illustrates both the integration operation and the dual of the integration operation in the double Brouwerian algebra $(DCS, \cup, \cap, \dot{-}, \div, G_1)$. The sets of programs in this example have the property that $DC(A)\{\, DC(Base) \,\}DC(B)$ is strictly less than $DC(A)[DC(Base)]DC(B)$.

program	**program**	**program**
$x := 0$;	$x := 0$;	$x := 0$;
$y := x$	$y := x$;	$y := x$;
end()	$z := y$	$z := y$
	end()	**end**(z).

This example illustrates a fundamental difference between integrating by the operation $a\,[base\,]b$ and integrating by $a\,\{\,base\,\}b$. With $a\,[base\,]b$ an insertion made in one integrand can "override" a deletion in the other integrand; in the example shown above, the insertion of statement $z := y$ in integrand $DC(B)$ overrides the deletion of $y := x$ from integrand $DC(A)$. By contrast, with $a\,\{\,base\,\}b$ a deletion in one integrand can override an insertion in the other integrand; in the example, the deletion of statement $y := x$ from integrand $DC(A)$ overrides the insertion of $z := y$ in integrand $DC(B)$. Consequently, we say that $a\,\{\,base\,\}b$ is the *deletion-preserving integration* of a and b with respect to *base*.

Remark. As a practical matter, the deletion-preserving integration operation is probably less important than the integration operation, since the emphasis in producing software is usually on enhanced functionality (*i.e.*, insertions) rather than on reduced functionality. Thus, ordinarily it is desirable that an element deleted as part of (say) programmer *A*'s change but needed for programmer *B*'s change (such as statement $y := x$ in the example given above) should appear in the integrated program. Only occasionally is it important to emphasize reduced functionality—for example when creating a specialized version of a system to run on a machine with a small address space.

5.4. Existence of a Maximum Compatible Integrand

This section re-examines the question of when there is an integrand compatible with a given base *base*, integrand *a*, and result *m*. In particular, we give a closed formula for the greatest solution of the equation $a\,[base\,]x = m$ and show that it is, in fact, the greatest solution. (Note that our formula for the greatest solution makes use of the quotient operation, and thus holds only for double Brouwerian algebras.) This result is then used to extend Theorem 4.12; Theorem 5.9 shows that, if they exist, the solutions to $a\,[base\,]x = m$ form a distributive lattice with a least element and a greatest element.

Definition 5.6. $x_{max} \triangleq m \sqcup (base \sqcap (m \div a))$.

Theorem 5.7. *If* $a\,[base\,]m = m$ *then* x_{max} *is the maximum* x *such that* $a\,[base\,]x = m$.

Proof.

Part I. Show that $a\,[base\,]x_{max} = m$.

The proof breaks into two parts: in part (i), we show that $a\,[base\,]x_{max} \sqsupseteq m$; in part (ii), we show that $a\,[base\,]x_{max} \sqsubseteq m$.

(i)

$$\begin{aligned} a\,[base\,]x_{max} &= a\,[base\,](m \sqcup (base \sqcap (m \div a))) \\ &= a\,[base\,]m \sqcup a\,[base\,](base \sqcap (m \div a)) && \text{by Proposition B.9} \\ &= m \sqcup a\,[base\,](base \sqcap (m \div a)) \\ &\sqsupseteq m && (*) \end{aligned}$$

(ii)

$$\begin{aligned}
a\,[base\,]x_{max} &= a\,[base\,](m \sqcup (base \sqcap (m \div a))) \\
&= a\,[base\,]m \sqcup a\,[base\,](base \sqcap (m \div a)) && \text{by Proposition B.9} \\
&= m \sqcup a\,[base\,](base \sqcap (m \div a)) \\
&\sqsubseteq m \sqcup (a\,[base\,]base \sqcap a\,[base\,](m \div a)) && \text{by Proposition B.10} \\
&= m \sqcup (a \sqcap a\,[base\,](m \div a)) && \text{by Proposition B.1} \\
&= m \sqcup (a \sqcap ((a \dot{-} base) \sqcup (a \sqcap base \sqcap (m \div a)) \sqcup ((m \div a) \dot{-} base))) \\
&= m \sqcup (a \sqcap (a \dot{-} base)) \sqcup (a \sqcap a \sqcap base \sqcap (m \div a)) \sqcup (a \sqcap ((m \div a) \dot{-} base))
\end{aligned}$$

Because $a \dot{-} base \sqsubseteq a$, the second term $(a \sqcap (a \dot{-} base))$ simplifies to $a \dot{-} base$. By the dual of Proposition A.12, $a \sqcap (m \div a) = a \sqcap m$, so the third term simplifies to $a \sqcap base \sqcap m$. However, $(a \dot{-} base) \sqcup (a \sqcap base \sqcap m) \sqsubseteq a\,[base\,]m = m$, so the last line in the above derivation simplifies to $m \sqcup (a \sqcap ((m \div a) \dot{-} base))$.

Additional simplification is possible because, by Proposition A.11, we have $(m \div a) \dot{-} base \sqsubseteq m \div a$. Therefore, $a \sqcap ((m \div a) \dot{-} base) \sqsubseteq m$, and $a\,[base\,]x_{max} \sqsubseteq m$. (**)

Combining (*) and (**), we have $m \sqsubseteq a\,[base\,]x_{max} \sqsubseteq m$; hence, $a\,[base\,]x_{max} = m$.

Part II. Show that x_{max} is the maximum x such that $a\,[base\,]x = m$.

Suppose that x is an element such that $a\,[base\,]x = m$. We will demonstrate that $x \sqsubseteq x_{max}$.

$$\begin{aligned}
m &= a\,[base\,]x \\
&= (a \dot{-} base) \sqcup (a \sqcap base \sqcap x) \sqcup (x \dot{-} base)
\end{aligned}$$

$$\begin{aligned}
&x \dot{-} base \sqsubseteq m \\
&\quad x \sqsubseteq m \sqcup base && (*)
\end{aligned}$$

$$\begin{aligned}
&a \sqcap base \sqcap x \sqsubseteq m \\
&\quad x \sqsubseteq m \div (a \sqcap base) && (**)
\end{aligned}$$

Putting (*) and (**) together, we have

$$\begin{aligned}
x &\sqsubseteq (m \sqcup base) \sqcap (m \div (a \sqcap base)) \\
&= (m \sqcap (m \div (a \sqcap base))) \sqcup (base \sqcap (m \div (a \sqcap base))) \\
&= m \sqcup (base \sqcap (m \div (a \sqcap base))) && \text{by the dual of Proposition A.13} \\
&= m \sqcup (base \sqcap ((base \sqcap m) \div (base \sqcap (a \sqcap base)))) && \text{by the dual of Proposition A.21} \\
&= m \sqcup (base \sqcap ((base \sqcap m) \div (base \sqcap a))) \\
&= m \sqcup (base \sqcap (m \div a)) && \text{by the dual of Proposition A.21} \\
&= x_{max}
\end{aligned}$$

□

Properties of Solutions of $a\,[base\,]x = m$

We can now extend the results from Section 4 concerning the properties of solutions of $a\,[base\,]x = m$.

Lemma 5.8. *Solutions of $a\,[base\,]x = m$ are closed under $\sqcup$.*

Proof. Let x_1 and x_2 be two solutions of $a\,[base\,]x = m$ (*i.e.*, $a\,[base\,]x_1 = m$ and $a\,[base\,]x_2 = m$).

$$\begin{aligned}
a\,[base\,](x_1 \sqcup x_2) &= a\,[base\,]x_1 \sqcup a\,[base\,]x_2 && \text{by Proposition B.9} \\
&= m \sqcup m \\
&= m
\end{aligned}$$

□

Theorem 5.9. *Solutions of* $a\,[base\,]x = m$ *form a distributive lattice with least element* x_{min} *and greatest element* x_{max}.

Proof. Immediate from Theorem 4.12, together with Theorem 5.7 and Lemma 5.8. □

Theorem 5.9 is illustrated in Figure 13.

6. Some Practical Considerations

6.1. Implementation

A program-integration tool that uses the HPR algorithm has been demonstrable since the summer of 1987 [24, 27]. With the integration tool, one is able to display program slices and integrate programs; if interference is detected during integration (*i.e.*, integration fails), the system provides an interactive facility to help the user diagnose the cause of interference [23].

The user interface for the integration tool incorporates a language-specific editor created using the Synthesizer Generator, a meta-system for creating interactive, language-based program-development systems [21]. The editor of the program-integration tool automatically supplies tags on program components (*i.e.*, assignment statements and predicates) so that common components can be identified in different versions. Data-flow analysis of programs is carried out according to the editor's defining attribute grammar and used to construct program dependence graphs. Commands added to the editor make use of these graphs to perform their actions. For example, the integration command invokes the integration algorithm on the program dependence graphs, reports whether the variant programs interfere, and, if there is no interference, builds the integrated program.

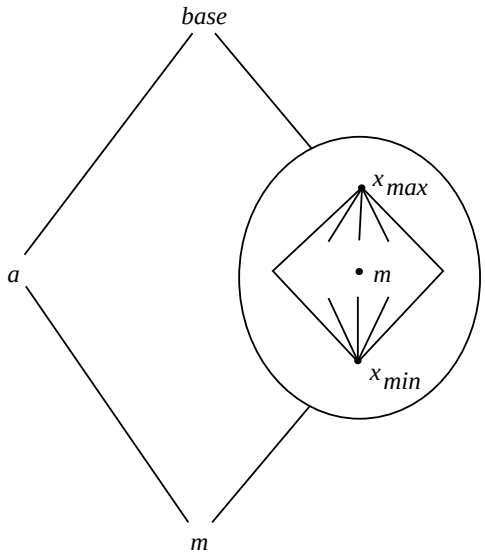


Figure 13. If solutions to the equation $a\,[base\,]x = m$ exist, then m itself is a solution, and the set of solutions forms a distributive lattice with least element x_{min} and greatest element x_{max}.

The implementation has recently been extended to incorporate the ideas described in this paper. The integration tool now also incorporates an editor and an interpreter for a higher-order functional language that operates on values of type Brouw, where a Brouw value is a downwards-closed set of (tagged) single-point slices. The primitive operations on Brouw values are the join, meet, and pseudo-difference operations of a Brouwerian algebra, together with a ternary operation for integration. Functional expressions are built up using lambda-abstraction, application, conditional expressions, let-clauses, and a least fixed-point operator. (Boolean and integer values are also provided.) A free variable in an expression (say x) denotes the Brouw created from the program in editing buffer x (*i.e.*, $DC(x)$). If no such buffer exists, the value is $\perp$, the least Brouw value. The editor displays a type for each expression; these types are supplied using the Milner algorithm for polymorphic type inference (algorithm *W*) [6, 17]. An evaluation command added to the editor invokes the interpreter on the expression, and—if the final result is a Brouw—builds the corresponding program (if one exists).

6.2. Integration Without Tags

Recall that in Section 2.2, we stated a requirement that a special program editor be used to create the program variants from the base program. Our assumption about this editor was that it provides a tagging capability so that common components can be identified in all versions. In the Brouwerian algebra (DCS, $\cup$, $\cap$, $\dot{-}$, G_1) defined Section 3.2, the elements are downwards-closed sets of *tagged* single-point slices; the tags on slice vertices are those supplied by the special program editor.

A different Brouwerian algebra that can be used for integrating programs is one whose elements are downwards-closed sets of *untagged* single-point slices. For this to work, we need a notion of *slice isomorphism*. For the purposes of defining such a notion it is convenient to add an additional label on flow dependence edges: if vertex v represents a program component that assigns to variable x, and x is the i^{th} operand of the program component represented by w, then the (set-valued) label on edge $v \longrightarrow_f w$ contains i. (For example, if x is used as both the i^{th} and j^{th} operands, then the label on edge $v \longrightarrow_f w$ is $\{ i, j \}$.)

Definition 6.1. Two single-point slices s_1 and s_2 are *isomorphic with respect to vertices* v_1 *and* v_2 iff all of the following hold:

(1) Slices s_1 and s_2 have the same number of vertices and the same number of edges.
(2) $(s_1 / v_1) = s_1$ and $(s_2 / v_2) = s_2$ both hold.
(3) There is a 1-to-1 and onto map M from the vertices of s_1 to the vertices of s_2, such that
  (i) $M(v_1) = v_2$, and
  (ii) For all vertices w of s_1, w and $M(w)$ are the same kind of vertex (*i.e.*, entry, assignment-statement, if-predicate, while-predicate, initial-definition, or final-use).
  (iii) For all vertices w of s_1, w and $M(w)$ have identical abstract syntax trees (*i.e.*, corresponding internal nodes of the two vertices' abstract syntax trees contain the same operator, and corresponding leaf nodes contain the same identifier or the same constant).
(4) For every edge $e = v \rightarrow w$ in s_1 there is an edge $e' = M(v) \rightarrow M(w)$ in s_2 such that:
  (i) The edge type of e (control, loop-independent flow, loop-carried flow, or def-order) is the same as the edge type of e';
  (ii) If e is a control dependence edge then its true/false label matches the true/false label of e';
  (iii) If e is a flow dependence edge then its operand-number label matches the operand-number label of e';
  (iv) If e is a loop-carried flow dependence edge with carrying-loop-predicate label p then the carrying-loop-predicate label of e' is $M(p)$;

(v) If e is a def-order edge with witness-vertex label u then the witness-vertex label of e' is $M(u)$.

What makes it plausible to use downwards-closed sets of untagged single-point slices for program integration is that, given slices s_1 and s_2 and vertices v_1 and v_2, it is possible to test in linear time whether s_1 and s_2 are isomorphic with respect to v_1 and v_2 [12]. Furthermore, by using hashing techniques the slice-set manipulations needed to perform operations in the algebra of downwards-closed untagged single-point slices can be performed in linear expected time (*i.e.* expected time linear in the sum of the sizes of the argument sets). The drawback of using untagged slice sets is that it entails additional costs for finding the program that corresponds to the set of dependence graphs that result from an integration.[13]

It must be noted that the two different definitions of slice-set algebras correspond to two different relatives of the HPR algorithm. In the algebra based on downwards-closed sets of untagged slices, if a program has multiple slices that are isomorphic, the corresponding slice set will have only one copy of the duplicated slice. For example, both of the following programs

```
program      program
  x := 0;      x := 0;
  y := x;      y := x;
  w := x       x := 0;
end()          w := x
             end()
```

have the following set of (untagged) slices:

```
{ program      program      program      program     }
| end()          x := 0       x := 0;      x := 0;   |
|                end()        y := x       w := x    |
|                             end()        end()     |
}
```

For this reason, the integration algorithm based on sets of untagged slices can produce a different answer than the algorithm based on sets of tagged slices (both in terms of the final program that is the result of an integration, as well as in the notion of when an integration fails due to interference). Nevertheless, for both algebras when the set resulting from an integration is feasible, the Integration Theorem (Theorem 2.9) holds. In other words, the same characterization of the execution behavior of the integrated program in terms of the execution behaviors of the base program and the two integrands applies to the HPR algorithm and to both slice-set algebras.

The benefits of doing without tags are two-fold. First, the class of integration problems that can be handled successfully (*i.e.*, without interference being reported) in the algebra based on sets of untagged slices is strictly larger than the class that can be handled in the algebra based on tagged slices. (The latter coincides with the class handled by the HPR algorithm.) Second, it is no longer necessary for program integration to be supported by a closed system; in principle, programs can be integrated even if they are created using ordinary text editors.

7. Relation to Previous Work

There has been previous work on merging functional programs [1], logic programs [15], and specifications [4]. Different models of integration have been used in each case. In Berzins's work on

[13]Defining a good heuristic for this program-reconstitution problem remains an open question.

integrating functional programs, variants *A* and *B* are merged without regard to *Base*. The functional program that results from the merge preserves the (entire) behavior of *both*; thus, *A* and *B* cannot be merged if they conflict at any point where both are defined. Similarly, Lakhotia and Sterling's 1–1 join operation is a two-way merge. However, in their work there is no notion of interference, and the characterization of the semantic properties of the merged program was left as an open question in [15]. Feather's work on integrating specifications does take *Base* into account, but although the integration algorithm preserves syntactic modifications, it does not guarantee any semantic properties of the integrated specification. Both the HPR algorithm and the algorithm introduced in Section 3 for integrating programs by combining downwards-closed sets of single-point slices are three-way integration operations that satisfy the semantic criterion stated in Section 1.

The notation $a\,[base\,]b$ that has been used here for the integration operation in Brouwerian algebras is taken from a paper by Hoare in which he investigated some of the properties of $a\,[base\,]b$ in Boolean algebras [7, 8]. However, nearly all of the questions examined in this work (for Brouwerian algebras) were not addressed by Hoare (for Boolean algebras).

In unpublished work, Susan Horwitz and I found proofs of several algebraic properties of the HPR algorithm. The results given in this work consist of the analogues for Brouwerian algebras of these earlier results, together with a number of new results. However, the method of proof used in this work is very different from the proof techniques used to establish these earlier results, which involved complicated arguments—with many sub-cases and argument by *reductio ad absurdum*—about operations on dependence graphs. In contrast, the proofs given in this work are *strictly algebraic* in nature, making use of the rich set of algebraic laws that hold in Brouwerian algebras.

The work described here was motivated by the desire to find a simpler way to prove properties about program integration. In this, I feel it succeeds—proofs in Brouwerian algebra are much less complicated than direct proofs about dependence graphs. It also provides a framework for studying common properties of program-integration algorithms. The integration operation in a Brouwerian algebra is defined purely in terms of $\sqcup$, $\sqcap$, and $\div$, and thus has an analogue in all Brouwerian algebras. Thus, to show that a proposed program-integration algorithm shares these properties, one merely has to show that the algorithm can be formulated as an integration operation in a Brouwerian algebra.

Acknowledgement

This work was greatly influenced by a series of conversations with Tony Hoare about the operation $a\,[base\,]b$ in Boolean algebras. His deft manipulations of expressions involving $a\,[base\,]b$ made clear the benefits of understanding the elegant algebraic laws of this ternary operator. He also pointed me to [7], which discusses properties of $a\,[base\,]b$ in Boolean algebras and articulates the advantages of the notation in which the *base* argument appears in the middle position.

Many of the questions examined in this paper were formulated jointly with Susan Horwitz in an earlier unpublished study of the algebraic properties of the HPR algorithm. In addition, she provided many comments and helpful suggestions as this paper was being prepared.

Appendix A: Algebraic Laws for Brouwerian Algebras

This appendix covers the algebraic laws that hold for Brouwerian algebras.[14] The material presented

[14]Propositions A.2–A.21 are taken from a list given in [20] (pp. 59-60). (In [20], the laws are expressed in dual form, using a "pseudo-complement" operator, rather than in the form given in Appendix A, where pseudo-difference is used.)

here makes the paper essentially self-contained. (Several of the easier proofs have been omitted and serve as simple exercises for the reader.) Not all of the propositions listed below are actually used in the paper; those not used have been included as additional background material. Further information about Brouwerian algebras can be found in [16] and [20].

In double Brouwerian algebras, the algebraic properties of $\div$ are dual to the ones listed below; given a property for $\dot{-}$, the corresponding property for $\div$ is obtained by making the following substitutions: $\sqcup$ for $\sqcap$, $\sqcap$ for $\sqcup$, $\div$ for $\dot{-}$, $\sqsubseteq$ for $\sqsupseteq$, $\sqsupseteq$ for $\sqsubseteq$, and "max" for "min."

Proposition A.1. $a \dot{-} b = \min\{ z \mid a \sqsubseteq b \sqcup z \}$.

Proof. The proof breaks into two parts: in part (i), we show that $a \dot{-} b \in \{ z \mid a \sqsubseteq b \sqcup z \}$; in part (ii), we show that, for all $w \in \{ z \mid a \sqsubseteq b \sqcup z \}$, $w \sqsupseteq a \dot{-} b$.

(i)

$a \dot{-} b \sqsubseteq a \dot{-} b$
$\quad a \sqsubseteq (a \dot{-} b) \sqcup b$ by Definition 3.5(iii)

Therefore, $a \dot{-} b \in \{ z \mid a \sqsubseteq b \sqcup z \}$.

(ii) Suppose $w \in \{ z \mid a \sqsubseteq b \sqcup z \}$.

$w \sqcup b \sqsupseteq a$
$\quad w \sqsupseteq a \dot{-} b$ by Definition 3.5(iii)

Therefore, $a \dot{-} b = \min\{ z \mid a \sqsubseteq b \sqcup z \}$. □

Proposition A.2. $b \dot{-} a = \bot$ *iff* $a \sqsupseteq b$.

Proof.

$\Rightarrow$ *case: Show that* $b \dot{-} a = \bot$ *implies* $a \sqsupseteq b$.

$b \dot{-} a \sqsubseteq \bot$
$\quad b \sqsubseteq a \sqcup \bot$ by Definition 3.5(iii)
$\quad\quad = a$

$\Leftarrow$ *case: Show that* $a \sqsupseteq b$ *implies* $b \dot{-} a = \bot$.

$\quad b \sqsubseteq a$
$\quad\quad = a \sqcup \bot$
$b \dot{-} a \sqsubseteq \bot$ by Definition 3.5(iii)

□

Proposition A.3. $a = b$ *iff* $b \dot{-} a = \bot = a \dot{-} b$.

Proof.

$\Rightarrow$ *case: Show that* $a = b$ *implies* $b \dot{-} a = \bot$ *and* $a \dot{-} b = \bot$.
$a \sqsupseteq b$ and $b \sqsupseteq a$; therefore, by Proposition A.2, $b \dot{-} a = \bot$ and $a \dot{-} b = \bot$.

$\Leftarrow$ *case: Show that* $b \dot{-} a = \bot$ *and* $a \dot{-} b = \bot$ *implies* $a = b$.
$b \dot{-} a = \bot$ and $a \dot{-} b = \bot$; therefore, by Proposition A.2, $a \sqsupseteq b$ and $b \sqsupseteq a$, which in turn implies $a = b$.
□

Proposition A.4. $a \dot{-} a = \bot$.

Proposition A.5. $\bot \dot{-} a = \bot$.

Proposition A.6. $b \dot{-} \bot = b$.

Proof.

(i)

$$\begin{aligned} b &\sqsubseteq b \sqcup \bot \\ b \dot{-} \bot &\sqsubseteq b \end{aligned}$$ by Definition 3.5(iii)

(ii)

$$\begin{aligned} b \dot{-} \bot &\sqsubseteq b \dot{-} \bot \\ b &\sqsubseteq (b \dot{-} \bot) \sqcup \bot \\ &= b \dot{-} \bot \end{aligned}$$ by Definition 3.5(iii)

Therefore, $b \dot{-} \bot = b$. □

Proposition A.7. $(a \dot{-} a) \sqcup b = b$.

Proposition A.8. $a \sqcup (b \dot{-} a) \sqsupseteq b$.

Proposition A.9. *If* $a_1 \sqsupseteq a_2$ *then* $b \dot{-} a_2 \sqsupseteq b \dot{-} a_1$.

Proof.

$$\begin{aligned} a_1 &\sqsupseteq a_2 && \text{by supposition} \\ a_1 \sqcup (b \dot{-} a_2) &\sqsupseteq a_2 \sqcup (b \dot{-} a_2) \\ &\sqsupseteq b && \text{by Proposition A.8} \end{aligned}$$

$a_1 \sqcup (b \dot{-} a_2) \sqsupseteq b$, hence, by Definition 3.5(iii), $b \dot{-} a_2 \sqsupseteq b \dot{-} a_1$. □

Proposition A.10. *If* $b_1 \sqsupseteq b_2$ *then* $b_1 \dot{-} a \sqsupseteq b_2 \dot{-} a$.

Proof.

$$\begin{aligned} a \sqcup (b_1 \dot{-} a) &\sqsupseteq b_1 && \text{by Proposition A.8} \\ &\sqsupseteq b_2 && \text{by supposition} \\ b_1 \dot{-} a &\sqsupseteq b_2 \dot{-} a && \text{by Definition 3.5(iii)} \end{aligned}$$

□

Proposition A.11. $b \sqsupseteq b \dot{-} a$.

Proposition A.12. $a \sqcup (b \dot{-} a) = a \sqcup b$.

Proof.

(i)

$$\begin{aligned} b \dot{-} a &\sqsubseteq b \\ a \sqcup (b \dot{-} a) &\sqsubseteq a \sqcup b \end{aligned}$$ by Proposition A.11

(ii)

$$\begin{aligned} a \sqcup (b \dot{-} a) &\sqsupseteq a \\ a \sqcup (b \dot{-} a) &\sqsupseteq b \\ a \sqcup (b \dot{-} a) &\sqsupseteq a \sqcup b \end{aligned}$$ by Proposition A.8

Therefore, $a \sqcup (b \dot{-} a) = a \sqcup b$. □

Proposition A.13. $(b \dot{-} a) \sqcup b = b$.

Proposition A.14. $(b \dot{-} a) \sqcup (c \dot{-} a) = (b \sqcup c) \dot{-} a$.

Proof.

(i)

$b \dot{-} a \sqsubseteq (b \sqcup c) \dot{-} a$ by Proposition A.10
$c \dot{-} a \sqsubseteq (b \sqcup c) \dot{-} a$ by Proposition A.10
$(b \dot{-} a) \sqcup (c \dot{-} a) \sqsubseteq (b \sqcup c) \dot{-} a$ (*)

(ii)

$b \sqcup c \sqsubseteq a \sqcup b \sqcup c$
$= a \sqcup (b \dot{-} a) \sqcup (c \dot{-} a)$ by Proposition A.12
$(b \sqcup c) \dot{-} a \sqsubseteq (b \dot{-} a) \sqcup (c \dot{-} a)$ (**)
by Definition 3.5(iii)

Putting (*) and (**) together, we have $(b \dot{-} a) \sqcup (c \dot{-} a) = (b \sqcup c) \dot{-} a$. □

Proposition A.15. $(c \dot{-} a) \sqcup (c \dot{-} b) = c \dot{-} (a \sqcap b)$.

Proof.

(i)

$c \dot{-} (a \sqcap b) \sqsupseteq c \dot{-} a$ by Proposition A.9
$c \dot{-} (a \sqcap b) \sqsupseteq c \dot{-} b$ by Proposition A.9
$c \dot{-} (a \sqcap b) \sqsupseteq (c \dot{-} a) \sqcup (c \dot{-} b)$ (*)

(ii)

$c \sqsubseteq (c \sqcup a) \sqcap (c \sqcup b)$
$= (c \sqcup a \sqcup (c \dot{-} b)) \sqcap ((c \dot{-} a) \sqcup c \sqcup b)$ by Proposition A.11
$= ((c \dot{-} a) \sqcup (c \dot{-} b) \sqcup a) \sqcap ((c \dot{-} a) \sqcup (c \dot{-} b) \sqcup b)$ by Proposition A.12
$= ((c \dot{-} a) \sqcup (c \dot{-} b)) \sqcup (a \sqcap b)$
$c \dot{-} (a \sqcap b) \sqsubseteq (c \dot{-} a) \sqcup (c \dot{-} b)$ (**)
by Definition 3.5(iii)

Putting (*) and (**) together, we have $c \dot{-} (a \sqcap b) = (c \dot{-} a) \sqcup (c \dot{-} b)$. □

Proposition A.16. $(c \dot{-} b) \dot{-} a = c \dot{-} (a \sqcup b) = (c \dot{-} a) \dot{-} b$.

Proof.

(i)

$c \sqsubseteq c \sqcup a \sqcup b$
$= (c \dot{-} (a \sqcup b)) \sqcup (a \sqcup b)$ by Proposition A.12
$c \dot{-} b \sqsubseteq (c \dot{-} (a \sqcup b)) \sqcup a$ by Definition 3.5(iii)
$(c \dot{-} b) \dot{-} a \sqsubseteq c \dot{-} (a \sqcup b)$ (*)
by Definition 3.5(iii)

(ii)

$c \sqsubseteq c \sqcup b \sqcup a$
$= (c \dot{-} b) \sqcup a \sqcup b$ by Proposition A.12
$= ((c \dot{-} b) \dot{-} a) \sqcup a \sqcup b$ by Proposition A.12
$c \dot{-} (a \sqcup b) \sqsubseteq (c \dot{-} b) \dot{-} a$ (**)
by Definition 3.5(iii)

Putting (*) and (**) together, we have $c \dot{-} (a \sqcup b) = (c \dot{-} b) \dot{-} a$. By symmetry, $c \dot{-} (a \sqcup b) = (c \dot{-} a) \dot{-} b$. □

Proposition A.17. $a \dot{-} c \sqsupseteq (b \dot{-} c) \dot{-} ((b \dot{-} a) \dot{-} c)$.

Proof.

$b \dot{-} c \sqsubseteq (a \sqcup b) \dot{-} c$ by Proposition A.10
$= ((b \dot{-} a) \sqcup a) \dot{-} c$ by Proposition A.12
$= ((b \dot{-} a) \dot{-} c) \sqcup (a \dot{-} c)$ by Proposition A.14

Therefore, by Definition 3.5(iii), $a \dot{-} c \sqsupseteq (b \dot{-} c) \dot{-} ((b \dot{-} a) \dot{-} c)$. □

Proposition A.18. $(b \dot{-} a) \sqcup (c \dot{-} b) \sqsupseteq c \dot{-} a$.

Proof.

$c \sqsubseteq a \sqcup b \sqcup c$
$= a \sqcup b \sqcup (c \dot{-} b)$ by Proposition A.12
$= a \sqcup (b \dot{-} a) \sqcup (c \dot{-} b)$ by Proposition A.12

Therefore, by Definition 3.5(iii), $(b \dot{-} a) \sqcup (c \dot{-} b) \sqsupseteq c \dot{-} a$. □

Proposition A.19. $(b \dot{-} a) \sqsupseteq (c \dot{-} a) \dot{-} (c \dot{-} b)$.

Proposition A.20. $a \sqsupseteq (a \sqcup b) \dot{-} b$.

Proof.

$(a \sqcup b) \dot{-} b = (a \dot{-} b) \sqcup (b \dot{-} b)$ by Proposition A.14
$= (a \dot{-} b) \sqcup \bot$ by Proposition A.4
$= a \dot{-} b$
$\sqsubseteq a$ by Proposition A.11

□

Proposition A.21. $c \sqcup ((c \sqcup b) \dot{-} (c \sqcup a)) = c \sqcup (b \dot{-} a)$.

Proof.

$c \sqcup ((c \sqcup b) \dot{-} (c \sqcup a)) = c \sqcup ((c \dot{-} (c \sqcup a)) \sqcup (b \dot{-} (c \sqcup a)))$ by Proposition A.14
$= c \sqcup (\bot \sqcup (b \dot{-} (c \sqcup a)))$ by Proposition A.2
$= c \sqcup (b \dot{-} (c \sqcup a))$
$= c \sqcup ((b \dot{-} a) \dot{-} c)$ by Proposition A.16
$= c \sqcup (b \dot{-} a)$ by Proposition A.12

□

Proposition A.22. $(a \dot{-} b) \dot{-} (a \sqcap b) = a \dot{-} b$.

Proof.

$(a \dot{-} b) \dot{-} (a \sqcap b) = ((a \dot{-} b) \dot{-} a) \sqcup ((a \dot{-} b) \dot{-} b)$ by Proposition A.15
$= ((a \dot{-} a) \dot{-} b) \sqcup (a \dot{-} (b \sqcup b))$ by Proposition A.16
$= (\bot \dot{-} b) \sqcup (a \dot{-} b)$ by Proposition A.4
$= \bot \sqcup (a \dot{-} b)$ by Proposition A.5
$= a \dot{-} b$

□

Proposition A.23. $(b \dot{-} a) \sqcup (b \sqcap a) = b$.

Proposition A.24. $(c \dot{-} b) \dot{-} a = (c \dot{-} a) \dot{-} (b \dot{-} a)$.

Proof.

(i)

$$\begin{aligned} a \sqcup (b \dot{-} a) \sqcup ((c \dot{-} b) \dot{-} a) &= a \sqcup b \sqcup (c \dot{-} b) && \text{by Proposition A.12} \\ &= a \sqcup b \sqcup c && \text{by Proposition A.12} \\ &\sqsupseteq c \\ c &\sqsubseteq a \sqcup (b \dot{-} a) \sqcup ((c \dot{-} b) \dot{-} a) \\ c \dot{-} a &\sqsubseteq (b \dot{-} a) \sqcup ((c \dot{-} b) \dot{-} a) && \text{by Definition 3.5(iii)} \\ (c \dot{-} a) \dot{-} (b \dot{-} a) &\sqsubseteq ((c \dot{-} b) \dot{-} a) && (*) \\ & && \text{by Definition 3.5(iii)} \end{aligned}$$

(ii)

$$\begin{aligned} b &\sqsupseteq b \dot{-} a && \text{by Proposition A.11} \\ (c \dot{-} a) \dot{-} (b \dot{-} a) &\sqsupseteq (c \dot{-} a) \dot{-} b && \text{by Proposition A.9} \\ &= (c \dot{-} b) \dot{-} a && (**) \\ & && \text{by Proposition A.16} \end{aligned}$$

Putting (*) and (**) together, we have $(c \dot{-} b) \dot{-} a \sqsupseteq (c \dot{-} a) \dot{-} (b \dot{-} a) \sqsupseteq (c \dot{-} b) \dot{-} a$; therefore, $(c \dot{-} b) \dot{-} a = (c \dot{-} a) \dot{-} (b \dot{-} a)$. □

Proposition A.25. $(a \sqcap b) \dot{-} c \sqsubseteq a \sqcap (b \dot{-} c)$.

Proof.

$$\begin{aligned} ((a \sqcap b) \dot{-} c) \dot{-} (a \sqcap (b \dot{-} c)) &= (((a \sqcap b) \dot{-} c) \dot{-} a) \sqcup (((a \sqcap b) \dot{-} c) \dot{-} (b \dot{-} c)) && \text{by Proposition A.15} \\ &= (((a \sqcap b) \dot{-} a) \dot{-} c) \sqcup ((a \sqcap b) \dot{-} (c \sqcup (b \dot{-} c))) && \text{by Proposition A.16} \\ &= (\bot \dot{-} c) \sqcup ((a \sqcap b) \dot{-} (b \sqcup c)) && \text{by Propositions A.2 and A.12} \\ &= \bot \sqcup \bot && \text{by Propositions A.5 and A.12} \\ &= \bot \end{aligned}$$

Therefore, by Proposition A.2, $(a \sqcap b) \dot{-} c \sqsubseteq a \sqcap (b \dot{-} c)$. □

Proposition A.26. $(a \sqcap b) \dot{-} c \sqsubseteq (a \dot{-} c) \sqcap (b \dot{-} c)$.

Proof.

$$\begin{aligned} ((a \sqcap b) \dot{-} c) &\sqsubseteq a \dot{-} c && \text{by Proposition A.10} \\ ((a \sqcap b) \dot{-} c) &\sqsubseteq b \dot{-} c && \text{by Proposition A.10} \end{aligned}$$

Therefore, $(a \sqcap b) \dot{-} c \sqsubseteq (a \dot{-} c) \sqcap (b \dot{-} c)$. □

Proposition A.27. $b \dot{-} (b \dot{-} a) \sqsubseteq a \sqcap b$.

Proof.

$$\begin{aligned} b &= (a \sqcap b) \sqcup (b \dot{-} a) && \text{by Proposition A.23} \\ b &\sqsubseteq (a \sqcap b) \sqcup (b \dot{-} a) \\ b \dot{-} (b \dot{-} a) &\sqsubseteq a \sqcap b && \text{by Definition 3.5(iii)} \end{aligned}$$

□

Proposition A.28. *If* $a \dot{-} b \sqsubseteq b$, *then* $a \sqsubseteq b$.

Proposition A.29. $c \dot{-} (a \sqcup b) \sqsubseteq (c \dot{-} a) \sqcap (c \dot{-} b)$.

Proof.

$$\begin{aligned} c \dot{-} (a \sqcup b) &\sqsubseteq (c \dot{-} a) && \text{by Proposition A.9} \\ c \dot{-} (a \sqcup b) &\sqsubseteq (c \dot{-} b) && \text{by Proposition A.9} \end{aligned}$$

Therefore, $c \dot{-} (a \sqcup b) \sqsubseteq (c \dot{-} a) \sqcap (c \dot{-} b)$. □

Proposition A.30. $(a \dot{-} b) \dot{-} (b \dot{-} a) = a \dot{-} b$.

Proof.

(i) $(a \dot{-} b) \dot{-} (b \dot{-} a) \sqsubseteq a \dot{-} b$ by Proposition A.11
(ii) $(a \dot{-} b) \dot{-} (b \dot{-} a) \sqsupseteq (a \dot{-} b) \dot{-} b$ by Propositions A.9 and A.11
$= a \dot{-} b$ by Proposition A.16

Therefore, $(a \dot{-} b) \dot{-} (b \dot{-} a) = a \dot{-} b$. □

Proposition A.31. $(a \sqcup b) \dot{-} (a \sqcap b) = (a \dot{-} b) \sqcup (b \dot{-} a)$.

Proposition A.32. $a \dot{-} (a \sqcap b) = a \dot{-} b$.

Appendix B: Algebraic Laws for the Integration Operation

This appendix gives proofs of the algebraic laws for the integration operation that were stated in Section 4.1.

Proposition B.1. $a\,[a\,]b = b$.

Proof.

$$\begin{aligned} a\,[a\,]b &= (a \dot{-} a) \sqcup (a \sqcap a \sqcap b) \sqcup (b \dot{-} a) \\ &= \bot \sqcup (a \sqcap b) \sqcup (b \dot{-} a) \\ &= b \qquad \text{by Proposition A.23} \end{aligned}$$

□

Proposition B.2. $a\,[base\,]a = a$.

Proof.

$$\begin{aligned} a\,[base\,]a &= (a \dot{-} base) \sqcup (a \sqcap base \sqcap a) \sqcup (a \dot{-} base) \\ &= (a \dot{-} base) \sqcup (a \sqcap base) \\ &= a \qquad \text{by Proposition A.23} \end{aligned}$$

□

Proposition B.3. $a\,[\bot]b = a \sqcup b$.

Proof.

$$\begin{aligned} a\,[\bot]b &= (a \dot{-} \bot) \sqcup (a \sqcap \bot \sqcap b) \sqcup (b \dot{-} \bot) \\ &= a \sqcup b \end{aligned}$$

□

Proposition B.4. $a\,[\top]b = a \sqcap b$.

Proof.

$$\begin{aligned} a\,[\top]b &= (a \dot{-} \top) \sqcup (a \sqcap \top \sqcap b) \sqcup (b \dot{-} \top) \\ &= \bot \sqcup (a \sqcap b) \sqcup \bot \\ &= a \sqcap b \end{aligned}$$

□

Proposition B.5. $a\,[a \sqcap b\,]b = a \sqcup b$.

Proof.

$$\begin{aligned} a\,[a \sqcap b\,]b &= (a \dot{-} (a \sqcap b)) \sqcup (a \sqcap a \sqcap b \sqcap b) \sqcup (b \dot{-} (a \sqcap b)) \\ &= ((a \sqcup b) \dot{-} (a \sqcap b)) \sqcup (a \sqcap b) \\ &= (a \sqcup b) \sqcup (a \sqcap b) \qquad \text{by Proposition A.12} \\ &= a \sqcup b \end{aligned}$$

□

Proposition B.6. $a\,[a \sqcup b\,]b = a \sqcap b$.

Proof.

$$\begin{aligned} a\,[a \sqcup b\,]b &= (a \dot{-} (a \sqcup b)) \sqcup (a \sqcap (a \sqcup b) \sqcap b) \sqcup (b \dot{-} (a \sqcup b)) \\ &= \bot \sqcup (a \sqcap b) \sqcup \bot && \text{by Proposition A.2} \\ &= a \sqcap b \end{aligned}$$

□

Proposition B.7. $a\,[base\,]\bot = a \dot{-} base$.

Proof.

$$\begin{aligned} a\,[base\,]\bot &= (a \dot{-} base) \sqcup (a \sqcap base \sqcap \bot) \sqcup (\bot \dot{-} base) \\ &= (a \dot{-} base) \sqcup \bot \sqcup \bot \\ &= a \dot{-} base \end{aligned}$$

□

Proposition B.8. $a\,[base\,]\top = a \sqcup (\top \dot{-} base)$.

Proof.

$$\begin{aligned} a\,[base\,]\top &= (a \dot{-} base) \sqcup (a \sqcap base \sqcap \top) \sqcup (\top \dot{-} base) \\ &= (a \dot{-} base) \sqcup (a \sqcap base) \sqcup (\top \dot{-} base) \\ &= a \sqcup (\top \dot{-} base) && \text{by Proposition A.23} \end{aligned}$$

□

Proposition B.9. $a\,[base\,](x_1 \sqcup x_2) = a\,[base\,]x_1 \sqcup a\,[base\,]x_2$.

Proof.

$$\begin{aligned} a\,[base\,](x_1 \sqcup x_2) &= (a \dot{-} base) \sqcup (a \sqcap base \sqcap (x_1 \sqcup x_2)) \sqcup ((x_1 \sqcup x_2) \dot{-} base) \\ &= (a \dot{-} base) \sqcup (a \sqcap base \sqcap x_1) \sqcup (a \sqcap base \sqcap x_2) \sqcup (x_1 \dot{-} base) \sqcup (x_2 \dot{-} base) \\ &\qquad \text{by Proposition A.14} \\ &= a\,[base\,]x_1 \sqcup a\,[base\,]x_2 \end{aligned}$$

□

Proposition B.10. $a\,[base\,](x_1 \sqcap x_2) \sqsubseteq a\,[base\,]x_1 \sqcap a\,[base\,]x_2$.

Proof. Because $a \sqcap base \sqcap (x_1 \sqcap x_2) \sqsubseteq a \sqcap base \sqcap x_1$ and $(x_1 \sqcap x_2) \dot{-} base \sqsubseteq x_1 \dot{-} base$, we have

$$\begin{aligned} a\,[base\,](x_1 \sqcap x_2) &= (a \dot{-} base) \sqcup (a \sqcap base \sqcap (x_1 \sqcap x_2)) \sqcup ((x_1 \sqcap x_2) \dot{-} base) \\ &\sqsubseteq (a \dot{-} base) \sqcup (a \sqcap base \sqcap x_1) \sqcup (x_1 \dot{-} base) \\ &= a\,[base\,]x_1 \end{aligned}$$

Therefore, $a\,[base\,](x_1 \sqcap x_2) \sqsubseteq a\,[base\,]x_1$. Similarly, we have $a\,[base\,](x_1 \sqcap x_2) \sqsubseteq a\,[base\,]x_2$. Consequently, $a\,[base\,](x_1 \sqcap x_2) \sqsubseteq a\,[base\,]x_1 \sqcap a\,[base\,]x_2$. □

Proposition B.11. *$a\,[base\,]b$ is monotonic in a.*

Proof. Suppose $a \sqsubseteq a'$. We will show that $a\,[base\,]b \sqsubseteq a'[base\,]b$.

$$\begin{aligned} a\,[base\,]b \dot{-} a'[base\,]b &= ((a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base)) \\ &\qquad \dot{-} ((a' \dot{-} base) \sqcup (a' \sqcap base \sqcap b) \sqcup (b \dot{-} base)) \\ &= ((a \dot{-} base) \dot{-} ((a' \dot{-} base) \sqcup (a' \sqcap base \sqcap b) \sqcup (b \dot{-} base))) \\ &\qquad \sqcup ((a \sqcap base \sqcap b) \dot{-} ((a' \dot{-} base) \sqcup (a' \sqcap base \sqcap b) \sqcup (b \dot{-} base))) \\ &\qquad \sqcup ((b \dot{-} base) \dot{-} ((a' \dot{-} base) \sqcup (a' \sqcap base \sqcap b) \sqcup (b \dot{-} base))) \\ &\qquad \text{by Proposition A.14} \\ &= (((a \dot{-} base) \dot{-} (a' \dot{-} base)) \dot{-} ((a' \sqcap base \sqcap b) \sqcup (b \dot{-} base))) \\ &\qquad \sqcup (((a \sqcap base \sqcap b) \dot{-} (a' \sqcap base \sqcap b)) \dot{-} ((a' \dot{-} base) \sqcup (b \dot{-} base))) \end{aligned}$$

$$\sqcup(((b \dot{-} base) \dot{-} (b \dot{-} base)) \dot{-} ((a' \dot{-} base) \sqcup (a' \sqcap base \sqcap b)))$$
by Proposition A.16

$$= \bot \sqcup \bot \sqcup \bot$$
$$= \bot$$

Therefore, by Proposition A.2, $a\,[base\,]b \sqsubseteq a'[base\,]b$. □

Proposition B.12. *$a\,[base\,]b$ is antimonotonic in base.*

Proof. Suppose $base \sqsubseteq base'$. We will show that $a\,[base']b \sqsubseteq a\,[base\,]b$.

$$a\,[base']b \dot{-} a\,[base\,]b = ((a \dot{-} base') \sqcup (a \sqcap base' \sqcap b) \sqcup (b \dot{-} base')) \dot{-} a\,[base\,]b$$
$$= ((a \dot{-} base') \dot{-} a\,[base\,]b) \sqcup ((a \sqcap base' \sqcap b) \dot{-} a\,[base\,]b)$$
$$\sqcup ((b \dot{-} base') \dot{-} a\,[base\,]b)$$ by Proposition A.14
$$= ((a \dot{-} base') \dot{-} ((a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base)))$$
$$\sqcup ((a \sqcap base' \sqcap b) \dot{-} ((a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base)))$$
$$\sqcup ((b \dot{-} base') \dot{-} ((a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base)))$$
$$= \bot \sqcup ((a \sqcap base' \sqcap b) \dot{-} ((a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base))) \sqcup \bot$$
by Propositions A.9 and A.2
$$= ((a \sqcap base' \sqcap b) \dot{-} (a \sqcap base \sqcap b)) \dot{-} ((a \sqcup b) \dot{-} base)$$ by Propositions A.16 and A.14
$$= (((a \sqcap base' \sqcap b) \dot{-} a) \sqcup ((a \sqcap base' \sqcap b) \dot{-} base) \sqcup ((a \sqcap base' \sqcap b) \dot{-} b))$$
$$\dot{-} ((a \sqcup b) \dot{-} base)$$ by Proposition A.15
$$= (\bot \sqcup ((a \sqcap base' \sqcap b) \dot{-} base) \sqcup \bot) \dot{-} ((a \sqcup b) \dot{-} base)$$ by Proposition A.2
$$= ((a \sqcap base' \sqcap b) \dot{-} base) \dot{-} ((a \sqcup b) \dot{-} base)$$
$$= (a \sqcap base' \sqcap b) \dot{-} (base \sqcup ((a \sqcup b) \dot{-} base))$$ by Proposition A.16
$$= (a \sqcap base' \sqcap b) \dot{-} (a \sqcup base \sqcup b)$$ by Proposition A.12
$$= \bot$$

Therefore, by Proposition A.2, $a\,[base']b \sqsubseteq a\,[base\,]b$. □

Proposition B.13. $a\,[x_1 \sqcup x_2]b \sqsubseteq a\,[x_1]b \sqcap a\,[x_2]b$.

Proof.

$a\,[x_1 \sqcup x_2]b \sqsubseteq a\,[x_1]b$ by monotonicity
$a\,[x_1 \sqcup x_2]b \sqsubseteq a\,[x_2]b$ by monotonicity

Therefore, $a\,[x_1 \sqcup x_2]b \sqsubseteq a\,[x_1]b \sqcap a\,[x_2]b$. □

Proposition B.14. $a\,[x_1 \sqcap x_2]b = a\,[x_1]b \sqcup a\,[x_2]b$.

Proof.

$$a\,[x_1 \sqcap x_2]b = (a \dot{-} (x_1 \sqcap x_2)) \sqcup (a \sqcap b) \sqcup ((b \dot{-} (x_1 \sqcap x_2)))$$ by Corollary 4.2
$$= (a \dot{-} x_1) \sqcup (a \dot{-} x_2) \sqcup (a \sqcap b) \sqcup (b \dot{-} x_1) \sqcup (b \dot{-} x_2)$$ by Proposition A.15
$$= a\,[x_1]b \sqcup a\,[x_2]b$$ by Corollary 4.2

□

Proposition B.15. $a\,[base\,]b \dot{-} base = (a \dot{-} base) \sqcup (b \dot{-} base)$.

Proof.

$$a\,[base\,]b \dot{-} base = ((a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base)) \dot{-} base$$
$$= ((a \dot{-} base) \dot{-} base) \sqcup ((a \sqcap base \sqcap b) \dot{-} base) \sqcup ((b \dot{-} base) \dot{-} base)$$
by Proposition A.14
$$= (a \dot{-} base) \sqcup \bot \sqcup (b \dot{-} base)$$ by Proposition A.2
$$= (a \dot{-} base) \sqcup (b \dot{-} base)$$

□

Proposition B.16. $a\,[base\,]b \dot{-} b = (a \dot{-} base) \dot{-} b$.

Proof.

$$\begin{aligned} a\,[base\,]b \dot{-} b &= ((a \dot{-} base) \sqcup (a \sqcap base \sqcap b) \sqcup (b \dot{-} base)) \dot{-} b \\ &= ((a \dot{-} base) \dot{-} b) \sqcup ((a \sqcap base \sqcap b) \dot{-} b) \sqcup ((b \dot{-} base) \dot{-} b) && \text{by Proposition A.14} \\ &= ((a \dot{-} base) \dot{-} b) \sqcup \bot \sqcup \bot && \text{by Proposition A.2} \\ &= (a \dot{-} base) \dot{-} b \end{aligned}$$

□

Appendix C: Relationship Between Pseudo-Difference and Quotient

This appendix gives a few laws that relate the operations $\sqcup$, $\sqcap$, $\dot{-}$, and $\div$ in a double Brouwerian algebra.

Proposition C.1. $(a \div b) \sqcap (b \dot{-} a) = a \sqcap (b \dot{-} a)$.

Proof. The proof splits into two cases: in part (i), we show that $(a \div b) \sqcap (b \dot{-} a) \sqsubseteq a \sqcap (b \dot{-} a)$; in part (ii), we show that $(a \div b) \sqcap (b \dot{-} a) \sqsupseteq a \sqcap (b \dot{-} a)$.

(i) We will show that $(a \div b) \sqcap (b \dot{-} a) \sqsubseteq a \sqcap (b \dot{-} a)$.

$(a \div b) \sqcap (b \dot{-} a) \sqsubseteq a \div b$, so $((a \div b) \sqcap (b \dot{-} a)) \sqcap b \sqsubseteq a$. (*)

$(a \div b) \sqcap (b \dot{-} a) \sqsubseteq b \dot{-} a \sqsubseteq b$, therefore $((a \div b) \sqcap (b \dot{-} a)) \sqcap b = (a \div b) \sqcap (b \dot{-} a)$. (**)

Substituting (**) into (*), we have

$(a \div b) \sqcap (b \dot{-} a) \sqsubseteq a$

Therefore, $(a \div b) \sqcap (b \dot{-} a) \sqsubseteq a \sqcap (b \dot{-} a)$. (†)

(ii) We will show that $(a \div b) \sqcap (b \dot{-} a) \sqsupseteq a \sqcap (b \dot{-} a)$ by showing that (a) $b \dot{-} a \sqsupseteq a \sqcap (b \dot{-} a)$, and (b) $a \div b \sqsupseteq a \sqcap (b \dot{-} a)$.

(a) follows immediately from the properties of $\sqcap$. To show (b), we have

$$\begin{aligned} a &\sqsupseteq a \\ &\sqsupseteq a \sqcap (b \dot{-} a) \\ &\sqsupseteq b \sqcap a \sqcap (b \dot{-} a) \end{aligned}$$

Therefore, $a \div b \sqsupseteq a \sqcap (b \dot{-} a)$.

Thus, $(a \div b) \sqcap (b \dot{-} a) \sqsupseteq a \sqcap (b \dot{-} a)$. (‡)

Combining (†) and (‡), we have $(a \div b) \sqcap (b \dot{-} a) = a \sqcap (b \dot{-} a)$. □

Proposition C.2. $b \sqcup ((b \div a) \dot{-} a) = b \div a$.

Proof. The proof splits into two cases: in part (i), we show that $b \sqcup ((b \div a) \dot{-} a) \sqsubseteq b \div a$; in part (ii), we show that $b \sqcup ((b \div a) \dot{-} a) \sqsupseteq b \div a$.

(i)

$$\begin{aligned} b &\sqsubseteq b \div a && \text{by the dual of Proposition A.11} \\ (b \div a) \dot{-} a &\sqsubseteq b \div a && \text{by Proposition A.11} \end{aligned}$$

Therefore, $b \sqcup ((b \div a) \dot{-} a) \sqsubseteq b \div a$. (*)

(ii)

$$\begin{aligned} b &\sqsupseteq a \sqcap (b \div a) && \text{by the dual of Proposition A.8} \\ &\sqsupseteq (b \div a) \dot{-} ((b \div a) \dot{-} a) && \text{by Proposition A.27} \end{aligned}$$

Therefore, $b \sqcup ((b \div a) \dot{-} a) \sqsupseteq b \div a$. (**)

Putting (*) and (**) together, we have $b \div a \sqsupseteq b \sqcup ((b \div a) \dot{-} a) \sqsupseteq b \div a$. Therefore, $b \sqcup ((b \div a) \dot{-} a) = b \div a$. □

Corollary C.3. $(b \div a) \dot{-} b \sqsubseteq (b \div a) \dot{-} a$.

Proof.

$b \div a = b \sqcup ((b \div a) \dot{-} a)$ by Proposition C.2

$b \div a \sqsubseteq b \sqcup ((b \div a) \dot{-} a)$

$(b \div a) \dot{-} b \sqsubseteq (b \div a) \dot{-} a$ by Definition 3.5(iii)

□

References

1. V. Berzins, "On merging software extensions," *Acta Informatica* **23** pp. 607-619 (1986).
2. H.B. Curry, *Foundations of Mathematical Logic*, Dover Publications, Inc., New York, NY (1977).
3. V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming environments based on structured editors: The MENTOR experience," pp. 128-140 in *Interactive Programming Environments*, ed. D. Barstow, E. Sandewall, and H. Shrobe,McGraw-Hill, New York, NY (1984).
4. M.S. Feather, "Detecting interference when merging specification evolutions," Unpublished report, Information Sciences Institute, University of Southern California, Marina del Rey, CA (1989).
5. J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.* **9**(3) pp. 319-349 (July 1987).
6. A.J. Field and P.G. Harrison, *Functional Programming*, Addison-Wesley, Reading, MA (1988).
7. C.A.R. Hoare, "A couple of novelties in the propositional calculus," *Zeitschr. f. math. Logik und Grundlagen d. Math.* **31** pp. 173-178 (1985).
8. C.A.R. Hoare, "A couple of novelties in the propositional calculus," pp. 325-331 in *Essays in Computing Science*, ed. C.B. Jones,Prentice-Hall, New York, NY (1989).
9. S. Horwitz, P. Pfeiffer, and T. Reps, "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* **24**(7) pp. 28-40 (July 1989).
10. S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).
11. S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).
12. S. Horwitz and T. Reps, "Efficient comparison of program slices," TR-982, Computer Sciences Department, University of Wisconsin, Madison, WI (December 1990).
13. D.J. Kuck, Y. Muraoka, and S.C. Chen, "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," *IEEE Trans. on Computers* **C-21**(12) pp. 1293-1310 (December 1972).
14. D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, "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).
15. A. Lakhotia and L. Sterling, "Composing recursive logic programs with clausal join," *New Generation Computing* **6**(2) pp. 211-225 (1988).
16. J.C.C. McKinsey and A. Tarski, "On closed elements in closure algebras," *Annals of Mathematics* **47**(1) pp. 122-162 (January 1946).
17. R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences* **17** pp. 348-375 (1978).
18. D. Notkin, R.J. Ellison, B.J. Staudt, G.E. Kaiser, E. Kant, A.N. Habermann, V. Ambriola, and C. Montangero, Special issue on the GANDALF project, *Journal of Systems and Software* **5**(2)(May 1985).
19. K.J. Ottenstein and L.M. Ottenstein, "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).

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

21. T. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, NY (1988).

22. T. Reps and W. Yang, "The semantics of program slicing," TR-777, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1988).

23. T. Reps and T. Bricker, "Illustrating interference in interfering versions of programs," *Proceedings of the Second International Workshop on Software Configuration Management*, (Princeton, NJ, Oct. 24-27, 1989), *ACM SIGSOFT Software Engineering Notes* **17**(7) pp. 46-55 (November 1989).

24. T. Reps, "Demonstration of a prototype tool for program integration," TR-819, Computer Sciences Department, University of Wisconsin, Madison, WI (January 1989).

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

26. T. Reps, "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).

27. T. Reps, "The Wisconsin program-integration system reference manual," Unpublished report, Computer Sciences Department, University of Wisconsin, Madison, WI (April 1990).

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