The Indigo Algorithm

Alan Borning, Richard Anderson, and Bjorn Freeman-Benson

Technical Report 96-05-01
Department of Computer Science and Engineering
University of Washington
July 1996

Authors' addresses:

Alan Borning and Richard Anderson
Dept. of Computer Science & Engineering
University of Washington
PO Box 352350
Seattle, Washington 98195 USA
{borning, anderson}@cs.washington.edu

Bjorn Freeman-Benson
Object Technology International Inc.
R. Buckminster Fuller Laboratory
201 - 506 Fort St.
Victoria, B.C. CANADA V8W 1E6
Bjorn_Freeman-Benson@oti.com

Abstract

Inequality constraints are useful for specifying various aspects of user interfaces, such as constraints that one window is to the left of another, or that an object is contained within a rectangle. However, current local propagation constraint solvers can't handle inequality constraints. We present Indigo, an efficient local propagation algorithm for satisfying acyclic constraint hierarchies, including inequality constraints.

A shorter version of this technical report will appear in the *Proceedings of the 1996 ACM Symposium on User Interface Software Technology.*

1 Introduction

Constraints are useful for a number of aspects of user interface construction, including layout, model-view consistency, and consistency of multiple views. In selecting the class of constraints to be supported in a UI toolkit, there are various tradeoffs between simplicity and power. Some of these tradeoffs include:

- one-way or multi-way constraints
- functional constraints only, or more general kinds of relations
- required constraints only, or constraint hierarchies

A constraint is *functional* if, for each of its constrained variables v not annotated as read-only, there is a unique value for v that will satisfy the constraint, given values for the other variables. A *constraint hierarchy* is a set of constraints labelled by strengths. The constraints labelled as "required" must be satisfied, while those labelled with weaker strengths are merely preferences [Borning et al. 92]. One important application of constraint hierarchies is in representing our desire that parts of a graphical object don't move unnecessarily, by placing weak stay constraints on them. This allows us to give a simple declarative semantics for constraint satisfaction in the presence of state and change over time. We can also use constraint hierarchies for expressing other kinds of preferences, for example, that a figure track the mouse if possible (but not if it bumps up against an immoveable object).

An important tradeoff concerning the constraint graph, rather than the individual constraints, is:

- acyclic constraint graphs only, or cycles allowed

All of these properties are declarative attributes of the constraints or the constraint graph, rather than of the constraint satisfaction algorithm. Naturally, however, the different properties place different requirements on the algorithm. One class of algorithm that has been explored by a number of researchers, and used in a variety of systems, is that of local propagation algorithms for multi-way constraints, constraint hierarchies, functional constraints only, and no cycles. These algorithms provide a good balance between expressiveness and efficiency; recent examples include DeltaBlue [Sannella et al. 93], SkyBlue [Sannella 94], and QuickPlan [Vander Zanden 96].

In local propagation algorithms, each constraint has a set of methods that can be used to satisfy the constraint. When a method is executed, it sets one of the constrained variables to a value such that the constraint is satisfied. All of the constraints in a traditional local propagation algorithm must be functional, since otherwise one couldn't provide the methods for that constraint.

Inequality constraints are useful in a variety of user interface applications, particularly in layout. For example, we might want a constraint that one window be to the left of another, or that a figure be contained within a rectangle. However, inequality constraints aren't functional, and aren't supported in traditional local propagation algorithms. Consider the constraint $a \leq b$: given b we can't uniquely determine a.

In previous UI systems, inequality constraints are either not provided, are only checked but not enforced, or have been supported using a variant of the simplex algorithm, iterative numerical methods, a backtracking technique, or the like. The research reported here was born out of a dissatisfaction with this state of affairs, and the conviction that, for an acyclic constraint network with inequalities, it should be possible to devise a straightforward local propagation algorithm.

Indigo is such an algorithm. The key idea in Indigo is that we propagate lower and upper bounds on variables (i.e. intervals), rather than specific values. We process the constraints strongest to weakest, tightening the bounds on variables as we go. When all the constraints have been processed, each variable will have a specific value — in other words, its upper bound will be equal to its lower bound.

Indigo is quite useful in its own right. However, it has actually been designed as a component of a larger algorithm, Ultraviolet, which partitions the constraint graph into different regions and uses a subsolver appropriate for the kind of constraints in that region. These subsolvers include two local propagation solvers: Blue for functional constraints over arbitrary kinds of objects and Indigo for inequality and other numeric constraints; and two cycle solvers: Purple for cycles of linear equality constraints and Deep Purple for linear inequality constraints. The subsolvers communicate via shared variables. An earlier version of Ultraviolet — which doesn't include Indigo — is described in [Borning & Freeman-Benson 95].

Ultraviolet has been implemented and tested, and Indigo has been tested separately as well as a standalone solver. The current version of Ultraviolet has also been used as the constraint solver in the Constraint Drawing Framework, a commercial graphics library written in Smalltalk. The Constraint Drawing Framework is an updated version of CoolDraw [Freeman-Benson 93]; the techniques for integrating the constraint mechanism with the other parts of the package are the same as descried in the earlier paper.

2 Related Work

Two systems designed for user interface applications that support inequality constraints are QOCA and DETAIL. QOCA [Helm et al. 92a] is a constraint solving toolkit that supports the incremental solution of simultaneous linear equality and inequality constraints while optimizing convex quadratic objective functions. Of particular note is its use of a parametric solved form, which allows solutions to be computed rapidly from changing input variables, and a projection algorithm that supports feedback regarding the possible range of movement of an object on the screen. However, QOCA doesn't support constraint hierarchies. QOCA has also been integrated with Unidraw to provide an architecture for constraint-based graphical editing [Helm et al. 92b]. DETAIL [Hosobe et al. 94] is an incremental solver for multi-way constraints and constraint hierarchies. It is more general than traditional local propagation, since it allows constraint cycles to be grouped into cells, which are then solved by an appropriate subsolver. The most recent version of DETAIL [Hosobe et al. 96] supports inequality constraints as well as functional constraints, although the current prototype has exponential time complexity.

There has been considerable work on interval constraints in other areas of computer science, particularly artificial intelligence and constraint logic programming. Davis [Davis 87] discusses the completeness and running time for interval propagation algorithms, as well as for other kinds of labels. Hyvönen [Hyvönen 92, Hyvönen et al. 94, Hyvönen 95] presents a number of interval constraint satisfaction algorithms, and also describes the generalization of interval propagation to division propagation. (A division is a union of ordered, non-overlapping intervals.) Another active area of research is the incorporation of interval constraints into constraint logic programming. Examples of such systems include CLP(BNR) [Older & Benhamou 93, Benhamou & Older 96] and Newton [Benhamou et al. 94]; see [Benhamou 95] for a recent survey. Most of these systems use local propagation to narrow the intervals that describe the permitted values for a variable (see for example the local tolerance propagation procedure in [Hyvönen 92]). These techniques are similar to that used in Indigo, although they apply only to required constraints rather than constraint hierarchies. (However, some of these systems also support the optimization of an objective function.)

3 Constraint Hierarchy Solutions

A solution to a constraint hierarchy is a mapping from variables to domain elements. Given a constraint hierarchy, if not all of the preferential constraints can be satisfied, we need a way to select which solutions are desired. In our previous work on DeltaBlue and SkyBlue we've used the locally-predicate-better comparator, in which we are concerned only whether or not a constraint is satisfied in a given solution. This comparator has proven quite satisfactory for functional constraints. However, for inequality constraints, an alternative comparator, locally-error-better, is superior. We give a brief, informal description of this comparator here; for a formal definition see reference [Borning et al. 92].

We will need to consider the error in satisfying a constraint. This error is 0 if and only if the constraint is satisfied, and becomes larger the further away the solution is from a satisfying one. For example, the error in satisfying the constraint $a = b$ for real numbers a and b is just $|a - b|$.

A solution S is locally-error-better if there is no other solution T that is better than S. Informally, T is better than S if there is some level k in the hierarchy such that the errors for all the constraints in levels 0 to $k - 1$ are exactly the same for T and S, and at level k the errors in satisfying each constraint using T are less than or equal to the errors using for S, and strictly less for at least one constraint. In general, there may be more than one locally-error-better solution to a given hierarchy.

To illustrate why locally-error-better gives more satisfactory results for inequality constraints in UI applications, consider an object constrained to lie within a fixed rectangle. Suppose the user is moving the object with the mouse, and tries to move it outside the rectangle. We'll make the mouse movement constraint strong but not required, so that the object will stop moving if it bumps up against an immovable obstacle, rather than giving an error. Using locally-predicate-better, if the user moves the object slowly, it will move as far as the side of the rectangle and then stop. However, if the user moves the mouse quickly (so that at one time the cursor is well within the rectangle and the next time outside), the figure will remain at the old location and not bump up against the side at all. (Since we can't satisfy the mouse constraint exactly, using locally-predicate-better we don't try to satisfy it at all.) Further, if the user tries to move the figure along the side of the rectangle it won't move unless the user gets the cursor positioned just on the boundary (but not outside it).

In contrast, with locally-error-better, the figure will follow the cursor until the cursor moves outside the rectangle. After that, the figure will move along the wall of the rectangle so that it is as close to the cursor as possible, as if the object were magnetically attracted to the cursor.

4 The Algorithm

In this section we describe the algorithm and the data structures it uses. For simplicity, we initially omit discussion of read-only variables; these are discussed at the end of this section. The input to the Indigo algorithm is a set of constraints, including both equalities and inequalities, and a set of variables. The algorithm finds a locally-error-better comparator to the constraints. (If there are multiple locally-error-better solutions, Indigo finds one of them.)

4.1 Principal Data Structures

Our implementation is in Smalltalk, and makes use of inheritance, so we'll describe it using object-oriented terminology. The two key kinds of objects are variables and constraints. Each constraint

represents a relation that should be satisfied. It has a strength (ranging from required to absolute weakest), and a set of variables to which it applies. A given collection of variables and constraints forms a constraint graph, which is assumed to be acyclic. (We can construct a *bipartite graph* by having a vertex for each variable and each constraint, and an edge from a constraint vertex to a variable vertex if the variable is constrained by the constraint [Gangnet & Rosenberg 92]. The constraint graph is said to be acyclic if the corresponding bipartite graph is acyclic.)

Each variable has a lower and an upper bound, which are initially $-\infty$ and $+\infty$ respectively. As with our previous algorithms DeltaBlue and SkyBlue, every variable has an implicit stay constraint at the absolute weakest strength. (A stay constraint is simply an equality constraint between a variable and its value prior to the start of constraint satisfaction; it expresses our desire that the values of variables shouldn't be changed unless necessary.) The existence of these stay constraints implies that when the algorithm terminates, each variable will have a specific value — in other words, its lower bound will be equal to its upper bound. This is a useful property for interactive graphical applications, in which objects need to be displayed at some specific place on the screen. The values found for the variables will constitute a locally-error-better solution to the constraints.

We make extensive use of intervals, which are objects that represent the lower and upper bounds on a variable. All intervals are closed intervals, unless the bound is infinite. We'll write $[a, b]$ to represent the closed interval consisting of all numbers between a and b, and $[a, \infty)$ to represent the half-open interval consisting of all numbers greater than or equal to a. We define the usual arithmetic operations ($+$, $-$, $\times$, $\div$) as messages to intervals. For example, $[l_1, u_1] \times [l_2, u_2]$ is defined to be the interval containing all numbers that are the product of one number from $[l_1, u_1]$ and another number from $[l_2, u_2]$. We also allow mixed-mode arithmetic, in which one operand is an interval and one is a number; in this case the numeric argument n is coerced to be the singleton interval $[n, n]$. Definitions of all the interval operations, including the case of infinite bounds, are given in Appendix A; see also reference [Alefeld & Herzberger 83]. This use of intervals, and in particular overloading the arithmetic operators for interval arithmetic and allowing infinite bounds, substantially simplifies both the presentation and implementation of the algorithm.

Finally we define the tighten message to variables (with an interval argument) to tighten the bounds on the variable if possible, given the argument. For example, if a variable currently has the bounds [3,10], if we tighten it using [5,8], its new bounds will be [5,8]. On the other hand, if a variable current has the bounds [3,10] and we tighten it using [50,100], its new bounds will be the singleton interval [10,10], since 10 is the number within the interval [3,10] that minimizes the distance to the interval [50,100]. The operation v.tighten(i) for a variable v and interval i is defined as follows:

```
if intersect(v.bounds,i) is not empty then
    v.bounds := intersect(v.bounds,i);
else if v.upperBound < i.lowerBound then
    v.lowerBound := v.upperBound;
else      /* we know v.lowerBound > i.upperBound */
    v.upperBound := v.lowerbound;
end if;
```

The tighten operation will be key in finding locally-error-better solutions, in which we minimize the error in satisfying non-required constraints, even when they can't be satisfied completely.

4.2 Propagation Methods

In a traditional local propagation solver, each constraint has a collection of methods. When a method is executed, it sets one of the constrained variables to a value such that the constraint is

satisfied. For example, the constraint $a+b=c$ has three methods: c := a+b, a := c-b, and b := c-a. In contrast, in Indigo, each constraint has a collection of bounds propagation methods. Thus, the $a+b=c$ constraint has three bounds propagation methods, which tighten the bounds on a, b, and c respectively:

```
a.tighten(c.bounds - b.bounds)
b.tighten(c.bounds - a.bounds)
c.tighten(a.bounds + b.bounds)
```

If we have previously tightened the bounds on say c, when we process the constraint $a+b=c$ we may then need to tighten the bounds on both a and b. This is in sharp contrast to the behavior of standard local propagation algorithms, in which to satisfy a constraint a single method is executed (and hence a single variable changed).

4.3 Algorithm Pseudo-Code

With these preliminaries out of the way, the description of the algorithm itself is straightforward. Informally, we process the constraints strongest to weakest. For each constraint cn, we try to satisfy cn as well as possible by tightening the bounds on the variables it constrains. (However, if cn is required, then we must be able to satisfy it exactly, or we signal an error.) Tightening the bounds on cn's variables may cause the bounds on other variables to be tightened, rippling out to further variables. To implement this, the algorithm keeps a queue of constraints to be checked. The queue contains just cn to start. If we can tighten the bounds on any of cn's variables, we add the other constraints on these variables to the queue. We keep processing the constraints on the queue until it is empty. After all the constraints have been processed (including the weakest stays), we will have determined values for all of the variables.

In pseudo-code, the algorithm is as follows.

```
all_constraints := list of all constraints, strongest first;
all_variables := set of all variables;
active_constraints := new set;
for v in all_variables do
    initialize v.bounds to unbounded;
end for;

for current_constraint in all_constraints do
    tight_variables := new set;
    queue := new queue;
    queue.add(current_constraint);
    while queue not empty do
        cn := queue.front;
        tighten_bounds(cn,queue,tight_variables,active_constraints);
        check_constraint(cn,active_constraints);
        queue.dequeue;
    end while;
end for;
```

The variable active_constraints holds a set of constraints that have already been considered, but which may need to be considered again if we subsequently tighten the bounds on one of their variables. During the processing of each constraint, queue holds a queue of constraints whose variables may

need to have their bounds tightened, and tight_variables is a set of variables whose bounds have been tightened while processing the current constraint. (When processing a given constraint, we never have to tighten the bounds on a variable twice, as will be discussed in Section 4.4. The set tight_variables is used to prevent the algorithm from reprocessing tight variables.)

The procedure tighten_bounds tightens the bounds on variables constrained by cn, and enqueues other affected constraints. The procedure check_constraint checks for unsatisfied required constraints, and also determines when constraints need to be added to or deleted from the set active_constraints.

```
procedure tighten_bounds(cn,queue,tight_variables,active_constraints)
    for v in cn.variables and v not in tight_variables do
        tighten_flag := cn.tighten_bounds_for(v);
        tight_variables.add(v);
        if tighten_flag then
            for c in v.constraints do
                if c in active_constraints and c not in queue then
                    queue.add(c);
                end if;
            end for;
        end if;
    end for;
end procedure tighten_bounds;
```

In the tighten_bounds procedure, the tighten_bounds_for message to the constraint cn tightens the bounds on its constrained variable v if possible, given the bounds on its other constrained variables. It returns true if the bounds were changed.

```
procedure check_constraint(cn,active_constraints)
    if cn is unary then
        if cn is required and cn is not satisfiable then
            exception(required_constraint_not_satisfied);
        end if;
        return;
    end if;
    if not all of c's variables have unique values then
        active_constraints.add(cn);
        return;
    end if;
    if cn is satisfied then
        active_constraints.delete(cn);
    else if cn is required then
        exception(required_constraint_not_satisfied);
    else exception(constraints_too_difficult);
    end if;
end procedure check_constraint;
```

In procedure check_constraint we first check whether the constraint cn is unary. (A unary constraint is a constraint on a single variable.) A unary constraint need only be processed once — there is never any reason to consider it again, since its influence is completely represented in the current bounds of the variable it constrains. Otherwise cn is n-ary. If not all of its variables have unique values, then we need to add cn to active_constraints, since we will need to consider cn again when the bounds on one of its variables are tightened. However, if all of cn's variables have unique values, cn need never be considered again, and hence can be deleted from active_constraints if it is there.

There are two exceptions that can be raised. If a required constraint can't be satisfied (i.e. we can't make its error be 0), we raise a required_constraint_not_satisfied exception. Finally, there is an assumption that all constraints in active_constraints can be satisfied completely. If this turns out to not be the case, we raise a constraints_too_difficult exception. The algorithm is thus sound but incomplete — that is, if it finds a solution, the solution will be correct, but there are some sets of constraints for which it won't find a solution. We could provide an algorithm that is both sound and complete at the cost of some additional complexity. However, the incompleteness arises only under unusual conditions, and so we haven't added these features. See Section 5.2 for more details.

4.4 Correctness

The idea for the proof is to show that the domains of the variables give exactly those values that can appear in a locally-error-better solution to the constraints. If each variable has a stay constraint on it, each domain will consist of a single value, and so we will have found a locally-error-better solution.

We have been describing the domains as intervals. However, in this section we generalize this and describe the domains as arbitrary sets. (There are cases where domains other than single intervals can arise — see Section 5.2.) We show that if D_i is the domain for variable v_i, then if $y \in D_i$ there is some locally-error-better solution where v_i takes on the value y. This global optimality property is a generalization of the global consistency property used in the constraint satisfaction and interval labelling literature [Hyvönen 92].

The proof demonstrates that the global optimality property is maintained as an invariant of the algorithm. Formally, the proof is an induction argument, where the induction is performed both on the order that the constraints are processed by the main loop of the algorithm and on the order that domains of variables are updated by tighten_bounds.

Before we proceed with the proof, we need to define the properties that the tighten operation must have so that the algorithm works correctly. The tighten operation restricts the domains of the variables of a constraint in a manner that does not exclude any correct solution, but at the same time only retains values that can participate in a correct solution.

Suppose $v_1, \ldots, v_k$ are the variables for constraint c with domains $D_1, \ldots, D_k$. Let $R = D_1 \times \cdots \times D_k$, and let $e_{\min}$ be the minimum error achievable for constraint c by any $x \in R$. ($e_{\min} = 0$ corresponds to satisfying the constraint.) Let $S = \{r \in R \mid r \text{ has error } e_{\min} \text{ for } c\}$. The tighten operation finds domains $D'_1, \ldots, D'_k$ with $D'_i = \pi_i S$. (π_i denotes the projection of a set of tuples onto its i-th component.)

We also need to define what it means to minimize the errors in satisfying a list of constraints. Consider a list of constraints $c_1, \ldots, c_t$. There will be some minimum achievable error e_1 for c_1. Let e_2 be the minimum achievable error for c_2 given that c_1 has error e_1, and so on. A solution that minimizes the errors in satisfying $c_1, \ldots, c_t$ is a solution for which c_i has error e_i for $1 \leq i \leq t$.

Proceeding now to the proof itself, we first show that any value in a domain can be extended to an optimal solution for the constraints processed so far. Next, we show that in an optimal solution, each variable takes on a value in its domain. Together, these lemmas allow us to prove that Indigo is correct.

Lemma 4.1 *Suppose constraints $c_1, \ldots, c_t$ have been processed. Suppose $y \in D_i$. There exists a solution to the constraints that minimizes the errors for $c_1, \ldots, c_t$ and in which variable v_i takes on the value y.*

Proof: The proof is by induction on t. The base case is for $t = 0$, which is trivial, since there are no constraints.

Suppose that the lemma holds when $t-1$ constraints have been processed, and now consider the case where $c_1, \ldots, c_t$ have been processed.

Consider the graph formed by the constraints $c_1, \ldots, c_t$. This graph is a forest. We root each tree by the constraint of highest index in the tree. Let T be the tree rooted at c_t. Each constraint c_j which is not a root has a variable v_i which is between c_j and the root of the tree. We call this variable the *leading variable* of the constraint. Root constraints do not have leading variables.

When the algorithm processes the constraint c_t, the tighten operation is applied to the constraints in T in a root to leaf order. (Our implementation uses breadth first search, although any other ordering consistent with the tree order would be fine, such as depth first ordering. In addition, we only consider those constraints in T that are in active_constraints. The algorithm would still be correct, although less efficient, if we considered all the constraints in T.)

Our first claim is that if the tighten operation is applied to a non-root constraint, it does not change the domain of the leading variable. Suppose tighten is applied to c_j with leading variable v_i. Let D_i be the domain of v_i, and let $\overline{D_i}$ be the previous domain of v_i, after processing c_{t-1} but before processing c_t. By the induction hypothesis, if $y \in \overline{D_i}$, then there is a solution with minimum error for c_j that has $v_i = y$. Since $D_i \subseteq \overline{D_i}$ every value in D_i can be extended to a minimum error solution of c_j. This means that the tighten operation will not shrink D_i when it is applied to c_j. This justifies the optimization of not tightening the variables in the set tight_variables.

We now show that if v_i is a variable in T and $y \in D_i$, then there is a solution where v_i takes on the value y. (If v_i is not a variable in T, then the induction hypothesis implies the result.) We prove this with an induction proof over the subtrees of T. We show that if A is a subtree of T rooted at a variable, then if v_i is a variable of A and $y \in D_i$, there is a solution with v_i taking a value y which satisfies all of the constraints of A. Our induction is on the height of A.

The base cases are trees of height 0 or 1. The tree of height 0 is an unconstrained variable, so it trivially satisfies the constraints. A tree of height one is a variable subject to a unary constraint c_j. When the constraint c_j is processed, D_i is restricted to the values satisfying c_j with the minimum possible error. As the domain D_i is further restricted, the values still satisfy c_j.

Suppose A is a tree rooted at v_l where v_l is the leading variable of constraints $c_{j_1}, \ldots, c_{j_k}$. There are three cases for v_i:

1. v_i is v_l. If we set v_i to y, then we can find assignments to the other variables of the constraints $c_{j_1}, \ldots, c_{j_k}$. Each of these variables is a root of a tree, so by induction we find an assignment to the variables of each the trees completing the assignment.

2. v_i is a non-leading variable of one of the constraints c_{j_s}. In this case, we find an assignment to the variables c_{j_s} which satisfies c_{j_s} with minimum error and has $v_i = y$. This fixes the value of v_l. We then can satisfy the other constraints containing v_l as in case 1. We now complete the assignment by induction.

3. v_i is some other variable of A. In this case, v_i is in a subtree A_{j_s} rooted at v_r which is in the constraint c_{j_s}. By induction, we find an assignment to the variables of A_{j_s}, which gives us an assignment to v_r. As in case 2, we can extend this to a full assignment.

To complete the proof, we need to show the result applies to the tree T. T is rooted at the constraint c_t with subtrees $A_1, \ldots, A_k$. Suppose that v_i is a variable of A_j, and A_j has root v_j. By induction,

we can find an assignment satisfying the constraints of A_j with $v_i = y$. This sets the variable v_j to some value. We now find values for the other variables of c_t which satisfy c_t with minimum error, and then find assignments to each subtree creating a complete assignment.

∎

The other direction of the proof is to show that all of the valid solutions are contained inside the domains.

Lemma 4.2 *Suppose there is a solution with variable $v_i = y$ that minimizes the errors for constraints $c_1, \ldots, c_t$. Then $y \in D_i$ after c_t has been processed.*

Proof:

The proof is by induction. The base case is for $t = 0$. This is trivial since variables are initialized to unbounded domains.

Suppose the result holds after $t-1$ constraints have been processed. We want to show that if there is a solution that satisfies the constraints with $v_i = y$, then $y \in D_i$. Let $\overline{D_i}$ denote the domain of v_i after processing c_{t-1} but before processing c_t.

Suppose v_i does not have its domain updated when c_t is processed, and there is a solution with $v_i = y$. By the inductive hypothesis, we must have $y \in \overline{D_i}$ and since $D_i = \overline{D_i}$, we have $y \in D_i$.

We now show that the lemma holds for the v_i's that are updated by giving an induction proof on the variables. This induction proof orders the variables in the order that they are updated by tighten.

The base case is for variables updated by applying tighten to the constraint c_t. Let c_t have variables $v_{i_1}, \ldots, v_{i_k}$ and consider an assignment that achieves the minimum errors for $c_1, \ldots, c_t$ with $v_{i_1} = y$. Since this assignment achieves the minimum errors for $c_1, \ldots, c_{t-1}$ we have $v_{i_j} \in \overline{D_{i_j}}$. The tighten routine will put v_{i_j} in D_{i_j}. Since domains do not shrink after they are first tightened (as shown in the proof for Lemma 4.1), we must have $y \in D_{i_1}$.

The inductive step is almost the same. Suppose that the constraint c_j has variables $v_{i_1}, \ldots, v_{i_k}$ and v_{i_k} is the leading variable. Consider an assignment that achieves the minimum errors for the constraints with $v_{i_1} = y$ and $v_{i_k} = z$. By the outer induction hypothesis, we have $v_{i_j} \in \overline{D_{i_j}}$ and by the inner induction hypothesis, we have $z \in D_{i_k}$. The tighten operation now guarantees that $y \in D_{i_1}$.

∎

Theorem 4.3 *Given an acyclic set of constraints, the Indigo algorithm computes a locally-error-better solution, assuming that the implementations of the* tighten *operation are correct.*

Proof:

Let S_L be a locally-error-better solution, and S_I a solution found by the Indigo algorithm. Assume that S_I is not a locally-error-better solution. Let c_l be the first constraint where S_L and S_I have different errors.

If S_L's error for c_l is greater than S_I's error for c_l, then S_L was not a locally-error-better solution.

Now suppose that S_I has a greater error for c_l than S_L had. Lemmas 4.1 and 4.2 show that the domains $D_1, \ldots, D_n$ include all the values which achieve the minimum errors for the constraints

$c_1, \ldots, c_{l-1}$. This means that the optimal solution for c_l is in the domains that are used by the tighten operation, contradicting our assumption that each constraint achieves the minimum possible error.

∎

4.5 Weak Algorithm

A weak version of the Indigo algorithm is one in which the tighten operation computes domains which may be *too big*. The weakened condition for tighten is that $D'_i \supseteq \pi_i S$. In this case we can still conclude that the solution found by the algorithm will be a locally-error-better one; but occasionally the weak version of the algorithm will be unable to find a solution (and will so indicate). In other words, the weak version of Indigo is sound but incomplete.

Lemma 4.4 *A weak version of Indigo computes a lower bound on the minimum error achievable for each constraint.*

Proof: Let $e'_1, \ldots, e'_m$ be the error bounds found by the weak version of the algorithm and $e_1, \ldots, e_m$ be the error bounds found by the exact algorithm. As long as $e'_i = e_i$, we must have $D_i{}' \supseteq D_i$ (where $D_i{}'$ is the domain computed by the weak algorithm). Let c_l be the first constraint on which the error bounds differ. In determining the minimum error for c_l, the weak algorithm is considering a set of tuples that includes all of the tuples considered by the exact algorithm. Therefore, the weak algorithm must find minimum bounds that are no greater than those that the exact algorithm finds. Hence, if $e' \neq e$, we must have $e' < e$. ∎

We can now show that the weak version of Indigo is sound but incomplete.

Theorem 4.5 *Given an acyclic set of constraints, which include stay constraints on every variable, a weak version of Indigo either computes a locally-error-better solution, or terminates with a* constraints_too_difficult *or a* required_constraint_not_satisfied *exception.*

Proof: We first show that the algorithm will terminate with a required_constraint_not_satisfied exception iff there is no solution that satisfies the required constraints. The algorithm processes the required constraints first, and each constraint will be submitted to the check_constraint procedure. If the constraint is unary, then we can determine immediately whether it is compatible with the current bounds for its variable. If it is compatible, we tighten the bounds (and so any further tightenings of the bounds for that variable will mean that the constraint is still satisfied). If it is not compatible, we raise a required_constraint_not_satisfied exception. If the constraint is n-ary and all its variables already have unique values, these are of course the values the variables will take in the returned solution. In this case we test whether or not the constraint is satisfied. If it is, it will be also satisfied in the returned solution; if it is not, we raise the required_constraint_not_satisfied exception.

Suppose instead the algorithm terminates successfully. By Lemma 4.4 we have found a lower bound on the minimum error achievable for each constraint. After we process a stay constraint, the bounds on a variable will always be tightened to a single value. Thus, since each variable has a stay constraint, we will have also found a specific value for each variable. Suppose this solution is not locally-error-better. Then there must be another solution that is better than the one found — but this contradicts the assertion that the algorithm finds lower bounds on the minimum achievable errors.

The only other possibility is that the algorithm terminates with a constraints_too_difficult exception. ∎

Our current implementation of Indigo is in fact a weak version of the algorithm, since our tighten operations can produce intervals that are too big (Appendix A). The advantage of doing this is that we can use a single interval as the bounds for each variable — otherwise we would need to support unions of intervals, which would be slower for multiplication and division constraints in some cases (Section 5.2).

4.6 No Solutions

If the required constraints cannot be satisfied, the algorithm signals a required_constraint_not_satisfied exception. Intuition might lead one to believe that if the required constraints can be satisfied there is always at least one locally-error-better solution, but it turns out this is not correct in certain pathological cases. Consider the following constraints:

required	$a > 0$
medium	$a = 0$

Even though clearly the required constraint can be satisfied, there are no locally-error-better solutions, since for any potential solution x that satisfies $a > 0$, $x/2$ has a smaller error for the $a = 0$ constraint. (See [Borning et al. 92] for further details.)

This problem doesn't arise in our current implementation, since we don't support strict inequalities. However, there would be no particular problem in handling this situation. Recall that the algorithm finds domains for the variables that contain exactly those values that can participate in a locally-error-better solution (or, for the weak version of the algorithm, supersets of the actual domains). Thus if the algorithm computes an empty domain, this implies that there are no solutions to the constraints, and we could raise an appropriate exception.

4.7 Performance

Suppose we have n variables, and m constraints. When a constraint is processed, we tighten bounds until they are all tight. The key observation is that the bounds on a variable are never tightened more than once when a constraint is processed. Because the constraint network is acyclic, if we encountered the same variable more than once, we would have a cycle. The run time for processing a single constraint is $O(n)$ in the worst case, so the algorithm has a run time bound of $O(nm)$. In most cases the algorithm will perform much better than this, although there do exist contrived examples which force the algorithm to perform this amount of work.

4.8 An Example

Consider the following constraints.

required	$a \geq 10$
required	$b \geq 20$
required	$a + b = c$
required	$c + 25 = d$
strong	$d \leq 100$
medium	$a = 50$
abs_weakest	$a = 5$
abs_weakest	$b = 5$
abs_weakest	$c = 100$
abs_weakest	$d = 200$

(The abs_weakest constraints are the absolute weakest stay constraints on a, b, c, and d, which have initial values of 5, 5, 100, and 200 respectively.)

We process the constraints strongest first. After processing $a \geq 10$, we tighten the bounds on a to $[10, \infty)$. We then process the $b \geq 20$ constraint, tightening b's bounds to $[20, \infty)$. Both of these constraints are unary, so they aren't added to active_constraints. Next, we process $a + b = c$, and so tighten the bounds on c to $[30, \infty)$. Since not all of this constraint's variables have unique values, we add $a + b = c$ to active_constraints. The last required constraint is $c + 25 = d$. We process it and tighten the bounds on d to $[55, \infty)$, and add $c + 25 = d$ to active_constraints as well.

We now go to the strong constraint $d \leq 100$. This tightens the bounds on d to [55,100]. This propagates to c through the $c + 25 = d$ constraint (which is in active_constraints), tightening c's bounds to [30,75]. We further propagate the change to both a and b through $a + b = c$, so that a's bounds are now [10,55] and b's bounds are [20,65]. Next, we process the medium constraint $a = 50$. Since 50 is within a's current bounds, we set a's bounds to [50,50], and tighten b's bounds to [20,25], c's bounds to [70,75], and d's bounds to [95,100]. Finally we process the absolute weakest stays. The stay on a has no effect, while the stay on b pins b's bounds to [20,20], since we try to satisfy $b = 5$ as well as possible given the current bounds. This propagates to c and then to d, giving bounds of [70,70] for c and [95,95] for d. The remaining stays have no effect. We have thus found the solution $a = 50$, $b = 20$, $c = 70$, $d = 95$, which is a locally-error-better solution to the hierarchy.

4.9 Read-Only Variables

One or more variables in a constraint cn may be annotated as read-only, so that cn won't affect their values [Borning et al. 92]. (We can thus simulate a one-way constraint system by annotating all but one of the variables as read-only in every constraint.) Read-only variables add no particular extra complexity to the algorithm. We partition the constraint graph into writeable regions, joined by variables annotated as read-only in one of the regions. We then perform a topological sort on the regions, putting a region in which a variable is annotated as read-only before the connected region in which it is writeable. We then solve the regions independently, in order. When we come to solve a region with a variable annotated as read-only, that variable will already have been given a value when solving the upstream region.

4.10 Locally-Predicate-Better Solutions

As discussed previously, the locally-error-better comparator gives more satisfactory results than locally-predicate-better. However, if we did want locally-predicate-better solutions, the algorithm can be easily modified to produce them.

To accomplish this, when we first process a constraint, we test whether it can be satisfied, given the

existing bounds on its variables. If it can, we proceed as in the locally-error-better algorithm. If it can't, we simply ignore the constraint, rather than trying to satisfy it as well as possible.

5 Future Work

5.1 Compilation

In many interactive applications, we need to solve the same constraint graph repeatedly with different input values for one or more of the variables, for example, when moving a point in a constrained figure with the mouse. We can achieve much better performance if we compile a plan for the given constraint graph, and repeatedly execute the plan for each input value. Fortunately, it is straightforward in Indigo to compile such plans. The compilation algorithm has been designed but not yet implemented.

To compile a plan, we first identify one or more of the variables as input variables to the plan. These are the variables that will be set to a different value each time the plan is executed, for example to the current x and y mouse positions. Essentially we then do a partial evaluation of the Indigo algorithm, producing a series of straight-line invocations of the tighten method.

To illustrate, let's modify the previous example by making a an input variable, with the input coming in at the medium strength.

required	$a \geq 10$
required	$b \geq 20$
required	$a + b = c$
required	$c + 25 = d$
strong	$d \leq 100$
medium	$a = input$
abs_weakest	$a = a.oldValue$
abs_weakest	$b = b.oldValue$
abs_weakest	$c = c.oldValue$
abs_weakest	$d = d.oldValue$

The compiled code produced for these constraints follows, interspersed with commentary. First we reset the bounds on a, b, c, and d to the bounds resulting from processing the constraints stronger than the input constraint $a = input$. (These bounds are independent of *input*.)

```
a.bounds := [10,55];
b.bounds := [20,65];
c.bounds := [30,75];
d.bounds := [55,100];
```

Next, we tighten the bounds on a to a single value, using *input*, and then call the propagation methods for b, c, and d. (Note that we need to call the propagation method for d after that for c.)

```
a.tighten(input);
b.tighten(c.bounds - a.bounds);
c.tighten(a.bounds + b.bounds);
d.tighten(c.bounds + 25);
```

At this point we can discard the absolute weakest stay on a, since we know a's bounds have been tightened to a single value. We then try to tighten b's bounds to satisfy the stay constraint on b, and propagate to c and d. (We know we don't need to propagate to a, since it already has a unique value.)

```
b.tighten(b.oldValue);
c.tighten(a.bounds + b.bounds);
d.tighten(c.bounds + 25);
```

After this is done, we also know that c and d have unique values, and so the remaining absolute weakest stays have no effect. Finally, note that we could inline the tighten messages, at the expense of larger code size.

5.2 Unions of Intervals

As noted in the algorithm pseudo-code presented in Section 4.3, under some conditions Indigo will not be able to solve a collection of constraints. The algorithm is thus sound but incomplete — if it finds an answer, the answer will be correct, but it won't always be able to find one. Here is an example of a hierarchy that can't be solved by Indigo.

required	$-1 \leq a \leq 1$
required	$-1 \leq b \leq 1$
strong	$a * b = 2$
weak	$a = 0.5$

In attempting to solve these constraints, we would first process $-1 \leq a \leq 1$ and $-1 \leq b \leq 1$, tightening the bounds on a and b to $[-1, 1]$. We then process the $a * b = 2$ constraint. We can't satisfy this constraint given the current bounds on a and b, but given our restriction to just storing single bounds no tightening is possible. We then process $a = 0.5$, setting a to 0.5, since this is within a's current bounds. We then propagate the restriction to b, tightening b's bounds to the singleton interval $[1, 1]$. At this point we detect that both a and b have unique values but the active constraint $a * b = 2$ can't be satisfied, so we raise a constraints_too_difficult exception.

Following Hyvönen [Hyvönen 92], we could avoid this problem by allowing the possible values for variables to be described by divisions (unions of ordered, non-overlapping intervals), rather than by single intervals. If this were done, we would also remove the test for constraints in active_constraints which end up being unsatisfied after specific values are found for their variables. (By using divisions instead of single intervals, for the standard arithmetic constraints we can completely represent the set of values for which the constraint attains its minimum error. The check guards against the situation where a constraint in active_constraints ends up not achieving its minimum error bound.)

In the example, after processing the $a * b = 2$ constraint, the divisions for a and b would both be $[-1, -1|1, 1]$, i.e. $[-1, -1] \cup [1, 1]$. Then, when we process the $a = 0.5$ constraint, we would restrict a to $[1, 1]$ (which satisfies the $a = 0.5$ constraint as well as possible). We would then propagate this restriction to b, giving $[1, 1]$ for b. This solution of $a = 1$, $b = 1$ is the single locally-error-better solution to the constraints.

There does not appear to be a practical need for this extension for our current collection of constraints ($+$, $-$, $\times$, $\div$, $=$, $\leq$, $\geq$) and intended applications, and processing divisions would be more expensive than processing single intervals. However, this extension would allow us to handle additional kinds of constraints such as absolute value, and so we plan to do an experimental implementation in the future.

5.3 Incrementality

Another important technique for making local propagation algorithms more efficient is to make them incremental. In an incremental algorithm such as DeltaBlue or SkyBlue, when a constraint is added or deleted, the current solution can be updated incrementally, rather than planning from scratch. Unfortunately, producing an incremental version of Indigo does not seem to be straightforward, and so far we have not been able to devise a satisfactory incremental version. However, we believe that the ability to do compilation will still result in quite adequate performance in our target interactive applications such as constraint-based drawing systems.

6 Acknowledgments

This project has been funded in part by Object Technology International and in part by the National Science Foundation under Grants IRI-9302249 and CCR-9402551. Thanks to Andy Montgomery, Gilles Trombettoni, Rony Shapiro, and the UIST referees for comments on drafts of this report.

References

[Alefeld & Herzberger 83] Gotz Alefeld and Jurgen Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983. Translation of: *Einfuhrung in die Intervallrechnung* (translated by Jon Rockne).

[Benhamou & Older 96] Frédéric Benhamou and William Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 1996. Forthcoming.

[Benhamou 95] Frédéric Benhamou. Interval constraint logic programming. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 1995.

[Benhamou et al. 94] Frédéric Benhamou, David McAllister, and Pascal Van Hentenryck. CLP(Intervals) revisited. In *International Symposium on Logic Programming*, pages 124–138, Ithaca, NY, USA, 1994. MIT Press.

[Borning & Freeman-Benson 95] Alan Borning and Bjorn Freeman-Benson. The OTI constraint solver: A constraint library for constructing interactive graphical user interfaces. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 624–628, Cassis, France, September 1995.

[Borning et al. 92] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[Davis 87] Ernest Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, July 1987.

[Freeman-Benson 93] Bjorn Freeman-Benson. Converting an existing user interface to use constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 207–215, Atlanta, Georgia, November 1993.

[Gangnet & Rosenberg 92] Michel Gangnet and Burton Rosenberg. Constraint programming and graph algorithms. In *Second International Symposium on Artificial Intelligence and Mathematics*, January 1992.

[Helm et al. 92a] Richard Helm, Tien Huynh, Catherine Lassez, and Kim Marriott. A linear constraint technology for interactive graphic systems. In *Graphics Interface '92*, pages 301–309, 1992.

[Helm et al. 92b] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Proceedings of the Third Eurographics Workshop on Object-oriented Graphics*, Champery, Switzerland, October 1992.

[Hosobe et al. 94] Hiroshi Hosobe, Ken Miyashita, Shin Takahashi, Satoshi Matsuoka, and Akinori Yonezawa. Locally simultaneous constraint satisfaction. In *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*, pages 51–62. Springer-Verlag LLNCS 874, 1994.

[Hosobe et al. 96] Hiroshi Hosobe, Satoshi Matsuoka, and Akinori Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, Boston, August 1996.

[Hyvönen 92] Eero Hyvönen. Constraint reasoning based on interval arithmetic: The tolerance propagation approach. *Artificial Intelligence*, 58(1–3):71–112, December 1992.

[Hyvönen 95] Eero Hyvönen. Evaluation of cascaded interval function constraints. In *Proceedings of the International Workshop on Constraint-Based Reasoning (CONSTRAINT-95)*, Melbourne Beach, Florida, April 1995.

[Hyvönen et al. 94] Eero Hyvönen, Stefano De Pascale, and Aarno Lehtola. Interval constraint programming in C++. In Brian Mayoh, Enn Tyugu, and Jaan Penjam, editors, *Constraint Programming*, pages 350–366. Springer-Verlag, 1994. NATO Advanced Science Institute Series, Series F: Computer and System Sciences, Vol. 131.

[Older & Benhamou 93] William Older and Frédéric Benhamou. Programming in CLP(BNR). In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 239–249, Newport, RI, USA, 1993.

[Sannella 94] Michael Sannella. SkyBlue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 1994 ACM Symposium on User Interface Software and Technology*, pages 137–146, 1994.

[Sannella et al. 93] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[Vander Zanden 96] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.

A Definitions of Interval Arithmetic Operations

As discussed in Section 4.1, we make extensive use of intervals in our implementation of Indigo. All intervals in our current implementation are closed intervals, unless the bound is infinite. Thus

$$\begin{aligned}
[l, u] &= \{x \mid l \leq x \leq u\} \\
[l, \infty) &= \{x \mid x \geq l\} \\
(-\infty, u] &= \{x \mid x \leq u\} \\
(-\infty, \infty) &= \Re
\end{aligned}$$

The arithmetic operations $+$, $-$, $\times$, $\div$ on intervals are defined as follows.

$$\begin{aligned}
A + B &= \{a + b \mid a \in A \wedge b \in B\} \\
A - B &= \{a - b \mid a \in A \wedge b \in B\} \\
A \times B &= \{a \times b \mid a \in A \wedge b \in B\} \\
A \div B &= \{a \div b \mid a \in A \wedge b \in B\}, \text{ if } 0 \notin [l_2, u_2]
\end{aligned}$$

It is straightforward to compute the results of these arithmetic operations for closed intervals with finite bounds, and, in the case of division, denominators that don't include 0. Here are equations for these cases.

$$\begin{aligned}
[l_1, u_1] + [l_2, u_2] &= [l_1 + l_2, u_1 + u_2] \\
[l_1, u_1] - [l_2, u_2] &= [l_1 - u_2, u_1 - l_2] \\
[l_1, u_1] \times [l_2, u_2] &= [l_3, u_3] \\
\text{where } l_3 &= \min(l_1 \times l_2, l_1 \times u_2, u_1 \times l_2, u_1 \times u_2) \\
u_3 &= \max(l_1 \times l_2, l_1 \times u_2, u_1 \times l_2, u_1 \times u_2) \\
[l_1, u_1] \div [l_2, u_2] &= [l_1, u_1] \times [1/u_2, 1/u_1], \text{ if } 0 \notin [l_2, u_2]
\end{aligned}$$

The propagation rules are more complex for the general case. If we wish to describe the bounds of variables exactly, we need to allow divisions (Section 5.2). For example, the propagation methods for $A \div B$ must exclude 0 from B (if it isn't already excluded).

In our current implementation, for efficiency we restrict the domains of variables to being single intervals. Therefore, in the following function definitions, we return the smallest single interval that includes the true result (Section 4.5).

We also support mixed-mode arithmetic, in which one operand is an interval and the other is a number — in this case, a number n can simply be coerced to be the interval $[n, n]$. In our implementation, in addition to coercing numbers to intervals as needed, we also include special-case code that handles the mixed-mode cases for multiplication and division to make these operations more efficient. For simplicity these cases are omitted from the function definitions below.

```
function + (a, b : Interval) : Interval
    real lower, upper;
    lower := (if a.lower = −∞ or b.lower = −∞ then −∞ else a.lower + b.lower);
    upper := (if a.upper = ∞ or b.upper = ∞ then ∞ else a.upper + b.upper);
    return Interval(lower,upper);
end function +;
```

```
function - (a, b : Interval) : Interval
    real lower, upper;
    lower := (if a.lower = −∞ or b.upper = ∞ then −∞ else a.lower - b.upper);
    upper := (if a.upper = ∞ or b.lower = −∞ then ∞ else a.upper - b.lower);
    return Interval(lower,upper);
end function -;

function * (a, b : Interval) : Interval
    real lower, upper, temp;
    boolean not_set;   /* not_set is true if the variables lower and upper haven't been initialized yet */
    not_set := true;
    if a.lower ≠ −∞ and b.lower ≠ −∞ then
        not_set := false;
        upper := lower := a.lower*b.lower;
    end if;
    if a.lower ≠ −∞ and b.upper ≠ ∞ then
        temp := a.lower*b.upper;
        lower := (if not_set then temp else min(lower,temp));
        upper := (if not_set then temp else max(upper,temp));
    end if;
    if a.upper ≠ ∞ and b.lower ≠ −∞ then
        temp := a.upper*b.lower;
        lower := (if not_set then temp else min(lower,temp));
        upper := (if not_set then temp else max(upper,temp));
    end if;
    if a.upper ≠ ∞ and b.upper ≠ ∞ then
        temp := a.upper*b.upper;
        lower := (if not_set then temp else min(lower,temp));
        upper := (if not_set then temp else max(upper,temp));
    end if;
    /* now check for an infinite lower bound for the result */
    if (a.lower = −∞ and (b.upper = ∞ or b.upper > 0))
        or (b.lower = −∞ and (a.upper = ∞ or a.upper > 0))
        or (a.upper = ∞ and (b.lower = −∞ or b.lower < 0))
        or (b.upper = ∞ and (a.lower = −∞ or a.lower < 0)) then
            lower := −∞;
    end if;
    /* check for an infinite upper bound for the result */
    if (a.lower = −∞ and (b.lower = −∞ or b.lower < 0))
        or (b.lower = −∞ and (a.lower = −∞ or a.lower < 0))
        or (a.upper = ∞ and (b.upper = ∞ or b.upper > 0))
        or (b.upper = ∞ and (a.upper = ∞ or a.upper > 0)) then
            upper := ∞;
    end if;
    return Interval(lower,upper);
end function *;
```

```
function / (n, d : Interval) : Interval
    real lower, upper, temp;
    boolean n_pos, n_neg, d_epsilon, d_neg_epsilon;    /* used later */
    boolean not_set;   /* not_set is true if the variables lower and upper haven't been initialized yet */
    not_set := true;
    if n.lower ≠ −∞ and d.lower ≠ −∞ then
        not_set := false;
        upper := lower := n.lower/d.lower;
    end if;
    if n.lower ≠ −∞ and d.upper ≠ ∞ then
        temp := n.lower/d.upper;
        lower := (if not_set then temp else min(lower,temp));
        upper := (if not_set then temp else max(upper,temp));
    end if;
    if n.upper ≠ ∞ and d.lower ≠ −∞ then
        temp := n.upper/d.lower;
        lower := (if not_set then temp else min(lower,temp));
        upper := (if not_set then temp else max(upper,temp));
    end if;
    if n.upper ≠ ∞ and d.upper ≠ ∞ then
        temp := n.upper/d.upper;
        lower := (if not_set then temp else min(lower,temp));
        upper := (if not_set then temp else max(upper,temp));
    end if;
    /* in checking for infinite lower or upper bounds we use the following booleans:
            n_pos : true if n includes some positive numbers
            n_neg : true if n includes some negative numbers
            d_epsilon : true if d includes positive numbers arbitrarily close to 0
            d_neg_epsilon : true if n includes negative numbers arbitrarily close to 0 */
    n_pos := (n.upper = ∞ or n.upper > 0);
    n_neg := (n.lower = −∞ or n.lower < 0);
    d_epsilon := ((d.lower = −∞ or d.lower ≤ 0) and (d.upper = ∞ or d.upper > 0));
    d_neg_epsilon := ((d.lower = −∞ or d.lower < 0) and (d.upper = ∞ or d.upper ≥ 0));
    /* check for an infinite lower bound for the result */
    if (n.lower = −∞ and (d.upper = ∞ or d.upper > 0))
        or (n.upper = ∞ and (d.lower = −∞ or d.lower < 0))
        or (n_neg and d_epsilon)
        or (n_pos and d_neg_epsilon) then
            lower := −∞;
    end if;
    /* check for an infinite upper bound for the result */
    if (n.lower = −∞ and (d.lower = −∞ or d.lower < 0))
        or (n.upper = ∞ and (d.upper = ∞ or d.upper > 0))
        or (n_neg and d_neg_epsilon)
        or (n_pos and d_epsilon) then
            upper := ∞;
    end if;
    return Interval(lower,upper);
end function /;
```