# Relational Constraint: A Fast Semantic Analysis Technique

Adam Brooks Webber*

TR 92-1319
December 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported by NSF grant IRI-8902721.

# Relational Constraint: A Fast Semantic Analysis Technique

Adam Brooks Webber*
Computer Science Department
Cornell University
Ithaca, NY 14853 USA
webber@cs.cornell.edu

December 1, 1992

**Abstract**

*Relational constraint* is a new method for fast semantic analysis of computer programs. It starts with a fixed, finite vocabulary of binary relations; for each data type this vocabulary forms a complete lattice. Relational constraint focuses on strengthening relations in this lattice, developing the strongest correct statement it can about the relation between any two objects in its universe. The relational constraint algorithm runs in $O(n^2)$ time. In this paper we discuss the motivation for this research, present the algorithm in detail, and show how it fits in the formal framework of abstract interpretation.

## Contents



*Supported by NSF grant IRI-8902721. 

# 1 Introduction

Thinner is an optimizer for a small language of purely functional, first-order programs. It was developed to demonstrate how the intuitive idea, "Don't make any unnecessary computations," can be formally understood and effectively automated. Like all advanced optimizers, Thinner requires semantic analysis of considerable power and speed. This paper describes a new inference technique called *relational constraint* used in Thinner's semantic analyzer.

## 1.1 Overview of relational constraint

Relational constraint is a fully-automatic technique with a forward-chaining flavor: it doesn't prove or disprove queries, it reports observations. It never makes incorrect observations but it does miss correct ones—its emphasis is definitely on speed, and it concludes its inference in $O(n^2)$ time.

The output of the relational constraint algorithm is a binary relation for every pair of objects of the same type. (Exactly what "objects" are under consideration is explained below.) The vocabulary of binary relations for a given data type is fixed. It forms a complete lattice in which the greatest lower bound of a set of relations yields a relation which is their conjunction, and the least upper bound of a set of relations yields a relation which is their disjunction. Relations are strengthened monotonically during inference: they start out at the top of the lattice (the relation which is true of all pairs) and may grow stronger during inference, but are never weakened or retracted. The vocabulary of binary relations for each type is described in detail in Section 2.

The leverage for strengthening relations comes from *constraints* on tuples of relations. A constraint $(P, S)$ gives a tuple $P$ of pairs of objects and a specification $S$ of allowable tuples of relations on those pairs. Constraints come in four flavors:

**Functional constraints.** Each operator in a program restricts the relations among its input and output values. In some cases (like the absolute value function) this restriction is a simple input-output relation which can be asserted immediately. In other cases (like subtraction) the relations among the input and output values, and relations with certain constants, are restricted in a more complex way. For example, if we have three objects $a$, $b$ and $c$ with $a =$ `(- b c)`, and if $b = c$, a functional constraint excludes the possibility that $c \neq 0$.

**Transitivity constraints.** For any three objects $a$, $b$ and $c$ of the same type, the relations among $a$, $b$ and $c$ are restricted by transitivity. For example, given that $a < b$ and $b < c$, the transitivity constraint for $a$, $b$ and $c$ excludes the possibility that $a \geq c$. We do not compute all possible transitivity constraints, but carefully limit transitivity to certain "interesting" cases.

**List property constraints.** List property constraints ensure that for each pair of list objects $(a, b)$ the relation between $a$ and $b$, the relation between $\mathrm{car}(a)$ and $\mathrm{car}(b)$, and the relation between $\mathrm{length}(a)$ and $\mathrm{length}(b)$, are as strong as possible taken jointly. For example, if we have two list objects $a$ and $b$ and two base-type objects $c$ and $d$, and if we know that $c = \mathrm{car}(a)$, $d = \mathrm{car}(b)$, and $a = b$, then the list property constraint for $a$ and $b$ excludes the possibility that $c \neq d$.

**Transparency constraints.** For each pair of calls to the same function, a transparency constraint insists that the relations reflect referential transparency. For example, if the program contains both $a = f(x)$ and $b = f(y)$, and if $x = y$, a transparency constraint excludes the possibility that $a \neq b$.

The nature of the specification of allowable tuples $S$ in a constraint $(P, S)$ and the method by which $S$ is used to strengthen the pairs in $P$ are described in Section 3. The full relational constraint algorithm is presented and analyzed in Section 4.

This paper is perhaps best thought of as an implementation note: it describes in detail an inference technique that we believe will be useful to others. There are implementation hints in Section 4.4 and two detailed examples in Section 5. Theorists are not neglected, however: Section 6 sketches a treatment of relational constraint in the formal framework of abstract interpretation.

In the remainder of this introductory section we touch briefly on Thinner itself, in just enough detail to give an idea of the inference problem the relational constraint method is intended to solve.

## 1.2 Thinner's language

The language that Thinner optimizes is a first-order, purely functional, strongly-typed language. There are six data types: boolean, bit vector (a 32-bit word), integer, and the three corresponding flat list types. The syntax of the language is generally Lisp-like, but for our purposes the syntax is irrelevant—by the time the semantic analyzer sees a program, it is already compiled into an intermediate form (described below). Functions take zero or more parameters and return one or more values. The operators of the language are shown in Table 1.

| Operator Type | Operator Names |
|---:|---|
| (integer $\times$ integer) $\longrightarrow$ integer | `+`, `-`, `*` |
| integer $\longrightarrow$ integer | `abs`, `-` |
| (integer $\times$ integer) $\longrightarrow$ boolean | `<`, `<=`, `>`, `>=`, `=`, `/=` |
| (boolean $\times$ boolean) $\longrightarrow$ boolean | `and`, `or`, `=`, `/=` |
| boolean $\longrightarrow$ boolean | `not` |
| (bit-vector $\times$ bit-vector) $\longrightarrow$ bit-vector | `and`, `or` |
| bit-vector $\longrightarrow$ bit-vector | `not` |
| (bit-vector $\times$ bit-vector) $\longrightarrow$ boolean | `=`, `/=` |
| (integer $\times$ integer-list) $\longrightarrow$ integer-list | `cons` |
| (boolean $\times$ boolean-list) $\longrightarrow$ boolean-list | `cons` |
| (bit-vector $\times$ bit-vector-list) $\longrightarrow$ bit-vector-list | `cons` |
| (integer-list $\times$ integer-list) $\longrightarrow$ integer-list | `append` |
| integer-list $\longrightarrow$ integer-list | `cdr` |
| (boolean-list $\times$ boolean-list) $\longrightarrow$ boolean-list | `append` |
| boolean-list $\longrightarrow$ boolean-list | `cdr` |
| (bit-vector-list $\times$ bit-vector-list) $\longrightarrow$ bit-vector-list | `append` |
| bit-vector-list $\longrightarrow$ bit-vector-list | `cdr` |
| integer-list $\longrightarrow$ integer | `car`, `length` |
| bit-vector-list $\longrightarrow$ integer | `length` |
| boolean-list $\longrightarrow$ integer | `length` |
| bit-vector-list $\longrightarrow$ bit-vector | `car` |
| boolean-list $\longrightarrow$ boolean | `car` |
| (integer-list $\times$ integer-list) $\longrightarrow$ boolean | `=`, `/=` |
| (boolean-list $\times$ boolean-list) $\longrightarrow$ boolean | `=`, `/=` |
| (bit-vector-list $\times$ bit-vector-list) $\longrightarrow$ boolean | `=`, `/=` |

Table 1: Operators and their types

As the table shows, several operators are overloaded: for example, `=` applies to any two objects of the same type. This overloading is resolved by the time the analyzer sees a program, but for simplicity we will continue to use the overloaded names. One important observation is that `car` does not work as in Common Lisp: it always returns a value of the base type, even when the list is empty. This language was chosen for the Thinner project because it allows the expression of complex recursive functions over several types, while skirting the issue of runtime exception handling.

## 1.3 The trace grammar representation

Thinner uses a special representation for the programs it tries to optimize. The basis of this representation is the *trace graph*, a directed acyclic graph structure that describes a single path of computation, a single mapping from inputs to outputs by way of intermediate values, without any loops or conditionals.

**Definition 1** *A* trace graph $G = (V, E)$ *consists of a finite vertex set* $V$ *and an edge set* $E$. *Each vertex* $v \in V$ *has a fixed* input arity *and a fixed* output arity, *which give the extents of its* input vector $v_{in}$ *and its* output vector $v_{out}$. *The vertex set* $V$ *is partitioned into five disjoint subsets:*

$V_i$: *a singleton with input arity 0 and output arity* $\geq 1$, *standing for the producer of a computation's inputs.*

$V_o$: *a singleton with input arity* $\geq 1$ *and output arity 0, standing for the consumer of a computation's outputs.*

$V_T$: *a set of vertices with input arity 1 and output arity 0. Each vertex in* $V_T$ *represents a true-or-false decision which, in this path of execution, gets the boolean value true on its input edge.*

$V_F$: *a set of vertices with input arity 1 and output arity 0. Each vertex in* $V_F$ *represents a true-or-false decision which, in this path of execution, gets the boolean value false on its input edge.*

$V_C$: *a set of vertices with input arity* $\geq 1$ *and output arity* $\geq 1$. *Each vertex in* $V_C$ *is assigned a deterministic function.*

*The edge set* $E$ *is an acyclic set of directed edges; each edge is either a pair* $(v_{out}(i), w_{in}(j))$, *representing a value produced by one vertex and required by another, or a pair* $(c, w_{in}(j))$, *representing a constant supplied to a vertex input. There is exactly one edge supplying every vertex input. Vertices and edges are treated as typed: every vertex input, vertex output, and constant belongs to a type, edges must be typed identically at both ends, and edges into* $V_T$ *or* $V_F$ *vertices must be of type boolean. For vertices in* $V_C$, *the type of the function determines the types of the vertex inputs and outputs.*

The vertices of a trace graph represent operators and the arcs represent intermediate values. Of course some of the computations in the course of a normal program's execution will be predicates: comparisons, tests, or other operators, each with a single true-or-false output. These predicates determine which one of the many paths of execution implicit in a program is actually followed. Any predicate evaluated in the course of a particular path of execution will appear in the trace graph for that path, but the output of that predicate will be delivered to a $V_T$ or $V_F$ vertex and so determined. A trace graph is like a partial evaluation of a program—a partial evaluation with respect to a fixed control path, as in the work of Perlin [Per89]. It has an execution only for those inputs that result in the proper predicate values.

The trace graph isn't powerful enough by itself to represent a program: a program may have infinitely many possible paths of execution, so we need some way to generate potentially infinite sets of trace graphs. Thinner uses a kind of graph grammar, the *trace grammar*, for this purpose. It is a specialization of the directed, node-label controlled (DNLC) graph grammars studied by Jannsens and Rozenberg [Roz87], not unlike the plex grammars of [BH90]. It works this way: a trace graph may contain vertices whose functions are non-terminals, and each non-terminal appears on the left-hand side of some production in the trace grammar. The right-hand side is a "daughter" trace graph whose input and output arities and types match those of the "mother" vertex. The host trace graph is transformed into a new graph by vertex replacement: the mother vertex is removed and the daughter graph is inserted in its place. The set of fully-terminal trace graphs that can be generated from the start graph by repetitions of this process is the set generated by the trace grammar.

For example, consider the following (exponential-time) implementation of the Fibonacci function:

```
(defun fib (x)
    (if (< x 2)
        x
        (+ (fib (- x 1)) (fib (- x 2)))))
```

Thinner compiles this program into the trace grammar shown in Figure 1. The grammar generates an infinite language of fully-terminal trace graphs. For each input $x$ to fib, this language contains the trace of (fib $x$). The trace grammar representation and its properties are discussed in more detail in [Web92].

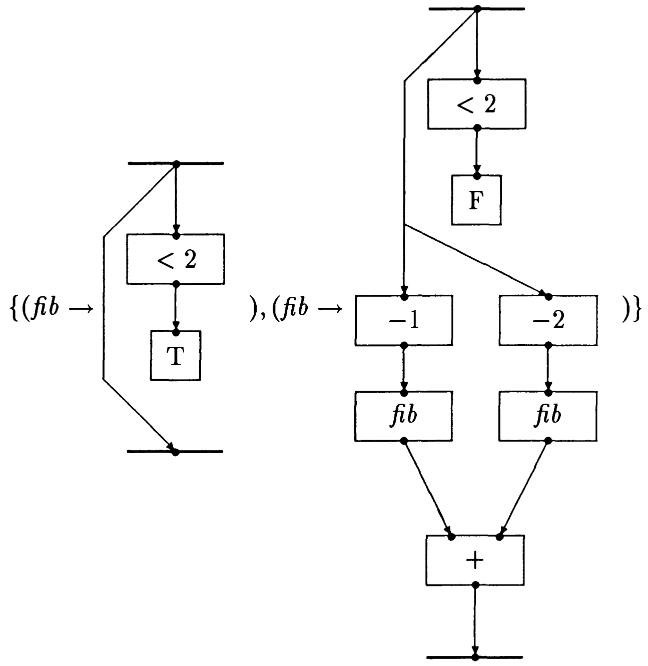


Figure 1: Trace grammar for the Fibonacci function

## 1.4 The inference needs of Thinner

Figure 2 shows the major components of Thinner. Thinner is a source-to-source optimizer: a *compiler* converts the input program into a trace grammar, and a *decompiler* converts the final trace grammar back into a program in the source language. In between, Thinner explores the trace grammar, looking for opportunities for optimization. It is an iterative process, and Thinner takes as much time as the user gives it: the more time it has, the more optimizations it may discover.

The grammar *transformer* works with a counterexample: a trace graph that has been shown to include unnecessary computation. (This trace graph, the "thinning example",

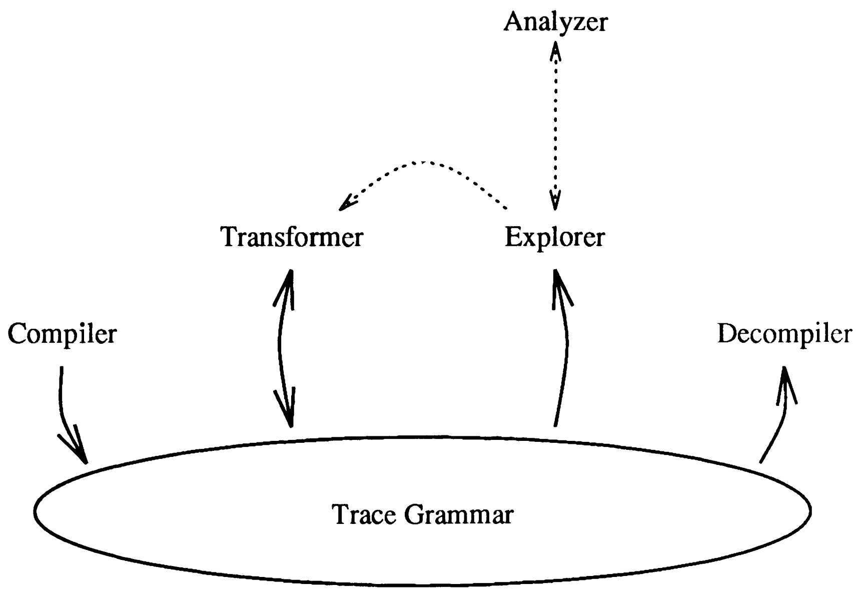


Figure 2: Structural overview of Thinner

may include non-terminals.) The transformer attempts to reformulate the grammar so that the language generated by the grammar never makes the mistake shown in the thinning example. (There is an interesting connection between this graph-grammar reformulation problem and a corresponding problem on context-free string grammars. This is explored in [Web].)

Where does the thinning example come from? A component called the *explorer* looks for unnecessary computations committed by the trace grammar. It looks at the right-hand-sides of productions in the grammar, and it unfolds longer and longer sequences of derivations. In effect, it examines individual partial paths through the program, embodying each such path in a trace graph.

The dotted lines in Figure 2 represent paths along which individual trace graphs are passed. The explorer calls on the analyzer to reason about individual trace graphs, to determine whether they contain any unnecessary computation—that is, to determine whether they compute redundant or unused values. This is the task for which relational constraint inference was developed: given a trace graph, find equivalences among values that occur during the computation the trace graph represents, as quickly as possible. The analyzer annotates the trace graph with equivalence information and returns it to the explorer. If there are suitable equivalences the graph is then passed to the transformer as a thinning example.

Note that Thinner is a "global" optimizer in this sense: the explorer examines interprocedural slices of the computation looking for thinning examples, and the transformer tries to correct all instances of a given example. But the analyzer reasons locally about each trace graph the explorer gives it.

The analyzer reasons about the relations between "objects" in a trace graph. Specifically, these objects are:

- all vertex outputs and constants from the graph; and

- the integer constants 0 and 1, the boolean constants false and true, the bit vector constant 00000000x0, and the three typed empty lists—whether they occur in the graph or not.

# 2 The vocabulary of binary relations

When other parts of Thinner propose a trace graph for semantic analysis, the analyzer's job is to find equivalences on objects in that graph. In fact, the relational constraint method finds a binary relation for every pair of objects in the graph; some of these may imply equivalence, but even those that do not provide leverage for relevant inference. The method of relational constraint will work for any vocabulary with suitable lattice properties; the vocabulary described here is the one we use in Thinner.

We begin by defining a small set $P^t$ of *primitive* binary relations for each data type $t$. The primitive relations for a type are exhaustive and mutually exclusive, so that for any constants $a$ and $b$ of type $t$, there is exactly one $r \in P^t$ for which $arb$. Further, every primitive relation has a reversed version, so that if $q \in P^t$, there is some $r \in P^t$ such that $arb$ if and only if $bqa$. ($q$ may be symmetric, in which case $r$ and $q$ will be identical.)

The primitive relations are used to build a lattice of *expressible* relations, which makes up the inference engine's vocabulary. We'll denote an expressible relation by indicating a set $R$ of primitive relations: we define $xRy$ as $\bigvee_{r \in R} xry$. So an expressible relation is just a set of primitive relations, at least one of which must be satisfied; each subset of the set of primitive relations gives a different expressible relation.

Because the primitive relations are exhaustive and mutually exclusive, this construction yields a lattice of expressible relations in which the disjunction of two relations $Q$ and $R$ is the least upper bound $(Q \cup R)$ and their conjunction is the greatest lower bound $(Q \cap R)$. The set of all primitives for the type, $P^t$, is the expressible relation which is true of all pairs of objects of type $t$, and the empty set is the expressible relation which is never true; we'll denote these relations $\top$ and $\bot$, respectively. For any expressible relation $Q$ there is a relation $R$ with the property that $xQy$ if and only if $\neg(xRy)$; $R$ is simply $P^t \setminus Q$. Finally, for any expressible relation $Q$ there is an expressible relation $R$ with the property that $xQy$ if and only if $yRx$; $R$ can be constructed by simply reversing every primitive in $Q$. (Again, $Q$ may be symmetric, in which case $Q$ and $R$ will be identical.)

By construction, our vocabulary of expressible relations has the following properties:

- If two relations are expressible, their conjunction and disjunction are expressible.
- If a relation is expressible, so is its negation.
- If a relation is expressible, the same relation with arguments reversed is also expressible.

There is an obvious bit-coding of the expressible relations for type $t$ (using one bit for each primitive relation in $P^t$). Using this representation disjunction can be computed using logical-or, conjunction using logical-and, and negation using logical-not.

## 2.1 Integers

Our primitive relations on integers $a$ and $b$ describe the sign of $a$: $a < 0$,$a = 0$, or $a > 0$; the sign of $b$: $b < 0$,$b = 0$, or $b > 0$; and the difference of the magnitudes $|a| - |b|$: $|a| - |b| \leq -2$,

$|a| - |b| = -1$, $|a| - |b| = 0$, $|a| - |b| = 1$, or $|a| - |b| \geq 2$. Not all combinations of these properties are consistent—for example, if both $a$ and $b$ are 0 the difference of their magnitudes can only be 0. But 29 of the combinations are; and these 29 make up our exhaustive set $P^{\text{int}}$ of mutually exclusive relations on integers. Table 2 shows the notation we use for these primitive relations.

| | $\lvert a\rvert - \lvert b\rvert \leq -2$ | $\lvert a\rvert - \lvert b\rvert = -1$ | $\lvert a\rvert = \lvert b\rvert$ | $\lvert a\rvert - \lvert b\rvert = 1$ | $\lvert a\rvert - \lvert b\rvert \geq 2$ |
|---|---|---|---|---|---|
| $a < 0, b < 0$ | --<2+ | --<1 | --=0 | -->1 | -->2+ |
| $a < 0, b = 0$ | | | | -0>1 | -0>2+ |
| $a < 0, b > 0$ | -+<2+ | -+<1 | -+=0 | -+>1 | -+>2+ |
| $a = 0, b < 0$ | 0-<2+ | 0-<1 | | | |
| $a = 0, b = 0$ | | | 00=0 | | |
| $a = 0, b > 0$ | 0+<2+ | 0+<1 | | | |
| $a > 0, b < 0$ | +-<2+ | +-<1 | +-=0 | +->1 | +->2+ |
| $a > 0, b = 0$ | | | | +0>1 | +0>2+ |
| $a > 0, b > 0$ | ++<2+ | ++<1 | ++=0 | ++>1 | ++>2+ |

Table 2: Notation for the primitive relations in $P^{\text{int}}$

There are $2^{29}$ subsets of $P^{\text{int}}$, giving the same number of expressible relations on integers. A variety of well-known relations on integers are among the expressible; Table 3 gives some examples.

| $xRy$, for $R =$ | Description |
|---|---|
| {--=0, 00=0, ++=0} | $x = y$ |
| {-->2+, -->1, -0>2+, -0>1, 0+<2+, 0+<1, ++<2+, ++<1} | $x < y$ |
| {--<1, 0-<1, +0>1, ++>1} | $x = \text{succ}(y)$ |
| {--=0, -+=0, 00=0, +-=0, ++=0} | $\lvert x\rvert = \lvert y\rvert$ |

Table 3: Examples of common expressible relations on integers

## 2.2 Booleans

There are four possible combinations of two booleans $x$ and $y$. This yields a set $P^{\text{bool}}$ of four primitive relations, which are denoted **ff**, **ft**, **tf**, and **tt**, with $x\{\mathtt{ft}\}y$ iff $x$ = false and $y$ = true, and so on.

There are sixteen subsets of the $P^{\text{bool}}$, giving the same number of expressible relations on booleans. These are described in Table 4.

## 2.3 Bit vectors

There are 15 non-empty sets of ordered pairs over the universe $\{0, 1\}$. Each primitive relation $r$ on bit vectors can be described by one of these sets: if $A$ is such a set, the corresponding primitive relation $r_A$ has $xr_Ay$ iff for every bit position, the ordered pair of bits drawn from

| $xRy$, for $R=$ | Description |
|---:|---|
| $\{\}$ | false for all $x$ and $y$ |
| $\{\texttt{ff}\}$ | $x=$ false and $y=$ false |
| $\{\texttt{ft}\}$ | $x=$ false and $y=$ true |
| $\{\texttt{tf}\}$ | $x=$ true and $y=$ false |
| $\{\texttt{tt}\}$ | $x=$ true and $y=$ true |
| $\{\texttt{ff},\texttt{ft}\}$ | $x=$ false |
| $\{\texttt{ff},\texttt{tf}\}$ | $y=$ false |
| $\{\texttt{ff},\texttt{tt}\}$ | $x=y$ |
| $\{\texttt{ft},\texttt{tf}\}$ | $x=\overline{y}$ |
| $\{\texttt{ft},\texttt{tt}\}$ | $y=$ true |
| $\{\texttt{tf},\texttt{tt}\}$ | $x=$ true |
| $\{\texttt{ft},\texttt{tf},\texttt{tt}\}$ | $\overline{y}\rightarrow x$ |
| $\{\texttt{ff},\texttt{tf},\texttt{tt}\}$ | $y\rightarrow x$ |
| $\{\texttt{ff},\texttt{ft},\texttt{tt}\}$ | $x\rightarrow y$ |
| $\{\texttt{ff},\texttt{ft},\texttt{tf}\}$ | $x\rightarrow\overline{y}$ |
| $\{\texttt{ff},\texttt{ft},\texttt{tf},\texttt{tt}\}$ | true for all $x$ and $y$ |

Table 4: The expressible relations on booleans

$x$ and $y$ at that position is in $A$, and every pair in $A$ occurs as the pair of bits drawn at some position. In other words:

$$xr_Ay \equiv \left(\bigwedge_{n\in[0..31]}\left(\bigvee_{(a,b)\in A} x_n = a \wedge y_n = b\right)\right) \wedge \left(\bigwedge_{(a,b)\in A}\left(\bigvee_{n\in[0..31]} x_n = a \wedge y_n = b\right)\right)$$

The set $P^{\text{bitv}}$ of primitive relations on bit vectors is the set of the fifteen such relations $r_A$. These primitive relations are exhaustive and mutually exclusive, as required, and a primitive relation $r_A$ can be reversed by reversing all the (boolean) primitive relations in $A$. We employ the notation shown in Table 5 for these primitive relations, but we stress that these relations are mutually exclusive. The relation `v=` is like equality, but by definition it excludes `v00` (where both are `00000000x0`) and `v11` (where both are `FFFFFFFFx0`).

There are $2^{15}$ subsets of $P^{\text{bitv}}$, giving the same number of expressible relations on bit vectors. Examples of expressible relations on bit vectors are given in Table 6.

## 2.4 Lists

There are actually three list types, but the primitive and expressible relations are the same for all three. These relations should be considered as applying only to lists of matching type.

Our primitive relations on lists $a$ and $b$ describe whether the shorter is a suffix of the longer; whether the two lists have the same car; and the difference of the lengths $|a|-|b|$: $|a|-|b|\leq -2$, $|a|-|b|=-1$, $|a|-|b|=0$, $|a|-|b|=1$, or $|a|-|b|\geq 2$. Not all combinations of these properties are consistent: if $a$ and $b$ have the same length and are suffixes of each other, they must have the same car. But 19 of the combinations are; and these 19 make up

| Notation | $ar_Ab$, for $A =$ |
|---|---|
| `v00` | $\{(0,0)\}$ |
| `v01` | $\{(0,1)\}$ |
| `v10` | $\{(1,0)\}$ |
| `v11` | $\{(1,1)\}$ |
| `v0x` | $\{(0,0),(0,1)\}$ |
| `vx0` | $\{(0,0),(1,0)\}$ |
| `v=` | $\{(0,0),(1,1)\}$ |
| `v=-` | $\{(0,1),(1,0)\}$ |
| `vx1` | $\{(0,1),(1,1)\}$ |
| `v1x` | $\{(1,0),(1,1)\}$ |
| `v>-` | $\{(0,1),(1,0),(1,1)\}$ |
| `v>` | $\{(0,0),(1,0),(1,1)\}$ |
| `v<` | $\{(0,0),(0,1),(1,1)\}$ |
| `v<-` | $\{(0,0),(0,1),(1,0)\}$ |
| `vxx` | $\{(0,0),(0,1),(1,0),(1,1)\}$ |

Table 5: The set $P^{\mathsf{bitv}}$ of primitive relations on bit vectors

| $xRy$, for $R =$ | Description |
|---|---|
| {`v00`} | $x = y = 0$ |
| {`v00`, `v11`, `v=`} | $x = y$ |
| {`v01`, `v10`, `v=-`} | $x = \overline{y}$ |
| {`v11`, `v=`} | $x = y \neq 0$ |
| {`v00`, `v01`, `v11`, `v0x`, `v=`, `vx1`, `v<`} | $x \vee y = y$ |

Table 6: Some common expressible relations on bit vectors

our exhaustive set $P^{\text{list}}$ of mutually exclusive relations on lists. Table 7 shows the notation we use for these primitive relations.

| | $\lvert a\rvert - \lvert b\rvert \leq -2$ | $\lvert a\rvert - \lvert b\rvert = -1$ | $\lvert a\rvert = \lvert b\rvert$ | $\lvert a\rvert - \lvert b\rvert = 1$ | $\lvert a\rvert - \lvert b\rvert \geq 2$ |
|---|---|---|---|---|---|
| suffix, same car | `sc<2+` | `sc<1` | `sc=0` | `sc>1` | `sc>2+` |
| suffix | `s<2+` | `s<1` | | `s>1` | `s>2+` |
| same car | `c<2+` | `c<1` | `c=0` | `c>1` | `c>2+` |
| | `<2+` | `<1` | `=0` | `>1` | `>2+` |

Table 7: The primitive relations in $P^{\text{list}}$

There are $2^{19}$ subsets of $P^{\text{list}}$, giving the same number of expressible relations on lists. Examples are given in Table 8, below.

| $xRy$, for $R =$ | Description |
|---|---|
| $\{\texttt{s<1}, \texttt{sc<1}\}$ | $x = \text{cdr}(y)$ |
| $\{\texttt{sc=0}, \texttt{c=0}, \texttt{=0}\}$ | $\lvert x\rvert = \lvert y\rvert$ |
| $X^{\text{list}} \setminus \{\texttt{sc=0}\}$ | $x \neq y$ |

Table 8: Some common expressible relations on lists

# 3 Finding strong relations

In the previous section we established a vocabulary of expressible relations, giving a complete lattice of relations on each type. Of course, every pair of identically-typed objects in a trace graph satisfies the top relation for that type; but our goal is to find the strongest relation we can, as quickly as possible.

## 3.1 The table of relations

The relational constraint algorithm uses a table of relations giving the strongest relation currently known for each pair of objects. This table is always relationally symmetric: the relation in the table for $(a, b)$ is always the reverse of the relation for $(b, a)$.

Most entries in the table of relations are initially the top relation for their type, indicating that we know nothing about the relation for that pair. But there are several exceptions:

- The relation of an object with itself is initially the equality relation. Note that the equality relation need not be maximally strong: for example, the equality relation on integers is $\{\texttt{--=0}, \texttt{00=0}, \texttt{++=0}\}$, but if the inference engine discovers that the object is in fact positive, this relation can be strengthened to $\{\texttt{++=0}\}$.

- The relation of an object with a constant can be initialized to something stronger than the top relation. For example, the relation between a list object $a$ and the empty list can be initialized to $\{\texttt{s>2+}, \texttt{sc>2+}, \texttt{s>1}, \texttt{sc>1}, \texttt{sc=0}\}$.

- The relation between two constants can be computed immediately. Exactly one primitive relation applies to any two constants.
- Some pairs have an initial relation established by a "single-relation operator" as described below.

## 3.2 Single-relation operators

Some of the operators in our language strengthen the relation on a single pair of objects. For example, suppose a trace graph includes the computation $a =$ `(-` $b$`)`. This establishes a relation between $a$ and $b$ that is expressible in the language of relations, so all we need do is strengthen the original relation between $a$ and $b$, $\top$, to the negation relation, {+-=0, -+=0, 00=0}. Table 9 shows the other operators in our language that can be treated this way.

| Operation | Relation on $(a, b)$ |
|---|---|
| $a =$ `(-` $b$`)` | {+-=0, -+=0, 00=0} |
| $a =$ `(abs` $b$`)` | {++=0, -+=0, 00=0} |
| $a =$ `(not` $b$`)`, $a$ and $b$ boolean | {ft, tf} |
| $a =$ `(not` $b$`)`, $a$ and $b$ bit vector | {v01, v10, v=-} |
| $a =$ `(cons` $c$ $b$`)` | {sc>1, s>1} |

Table 9: Operators that can be captured in a single relation

In addition, vertices in $V_T$ strengthen the relation between the input object and itself from {tt, ff} to {tt}, and vertices in $V_F$ strengthen the relation to {ff}. So we treat these vertices as single-relation operators too.

## 3.3 The constraint computation

Suppose there is an addition $a =$ `(+` $b$ $c$`)` in the trace graph. The relations among $a$, $b$ and $c$ can't be captured by directly strengthening entries in the table of relations. But because $c$ is the sum of $a$ and $b$, not all of the $29^3$ triples of primitive integer relations for the pairs $((a, b), (b, c), (a, c))$ are possible. For example, if $b$ and $c$ are both positive, neither can be greater than $a$; so tuples like ({++>2+}, {++>2+}, {++<2+}) can be excluded. Of course, if the inference engine ever developed exactly those relations for $a$, $b$ and $c$ it would have a contradiction. More significantly, if it ever develops a triple like ({++>2+}, {++>2+}, {++<2+, ++>2+}) it can strengthen it to ({++>2+}, {++>2+}, {++>2+}) just by eliminating the impossible.

**Definition 2** *Let $T$ be any $k$-tuple of types. A* constraint *$(P, S)$ consists of a $k$-tuple $P$ of pairs of objects, in which the two objects in each pair $P_i$ are of type $T_i$; and a* constraint set *$S$ of $k$-tuples of expressible relations, in which each relation $C_i$ for $C \in S$ is of type $T_i$.*

For any $k$-tuple of types $T$ consider the space of $k$-tuples of primitive relations on the corresponding types:

$$T\text{-space} = X^{T_1} \times \ldots \times X^{T_k}$$

Any tuple $Q$ of *expressible* relations describes a cube in $T$-space, for the appropriate tuple of types $T$:

$$\{(r_1, \ldots, r_k) \mid \bigwedge_{i \in [1..k]} \bigvee_{q \in Q_i} r_i = q\}$$

A constraint set $S$ is a set of tuples of relations: it describes the region of $T$-space which is the disjunction of the cubes described by each $C \in S$. The inference procedure takes the current tuple of relations $Q$ for the pairs in $P$ (remember, this describes a cube in the tuple space) and the constraint set $S$, and yields the smallest cube $R$ containing the conjunction of $Q$ and $S$. Each pair in $R$ is at least as strong as the corresponding pair in $Q$; if it's stronger, we've gained information about a relation.

**function** Constrain($Q, S$);
  $Q$ is a tuple of relations;
  $S$ is a set of tuples of relations;
  Constrain($Q, S$) returns a tuple of relations;
**begin**
  $R :=$ the tuple of bottom relations;
  **for** each tuple $C$ in $S$
    **begin**
      $I := Q \wedge C$ (the componentwise conjunction);
      **if** $I$ does not contain a bottom relation **then**
        $R := R \vee I$ (the componentwise disjunction)
    **end**;
  **return** $R$
**end**;

## 3.4 Constraint sets

Now we have a method for constraining a tuple of relations given a constraint set. The constraint set describes a space of legal tuples of primitives; where does it come from? Each constraint set used by Thinner is generated by a program (and tested by another program); almost all the constraint sets are defined before Thinner runs. This section describes Thinner's constraint sets. Most have less than 10 elements, and the largest (the list transitivity constraint set) only 300. (Compare this to the size of the tuple spaces, which contain as many as $29^6$ elements).

We use four types of constraints: functional constraints, transitivity constraints, list property constraints, and transparency constraints.

### 3.4.1 Functional constraints

Some of the operators in our language fix a single binary relation, as described previously; others determine a restriction on the relations among the input and output objects for a vertex. For each such operator in the language there is a fixed constraint set $S$; for each such vertex in a trace graph we establish a constraint $(P, S)$ that connects the appropriate tuple $P$ of object pairs to the constraint set for that operator.

**Integers** An addition operation $a = (+\ b\ c)$ and the successor and predecessor relations are intimately related. If we learn that $|c| = 1$, for example, we want to eliminate many of the possible primitive relations between $a$ and $b$; and if we learn that $a$ is the successor of $b$, we want to be able to conclude that $c = 1$. To make this kind of inference possible an addition constraint $(P, S)$ doesn't have $P = ((a,b),(b,c),(a,c))$, as you might expect; instead $P = ((a,b),(b,c),(a,c),(a,1),(b,1),(c,1))$.

The constraint set for a subtraction operation $b =$ `(- a c)` also contains sextuples—in fact, it's the same constraint set as for addition (but with the pairs treated in a different order), since since $b =$ `(- a c)` if and only if $a =$ `(+ b c)`. The constraint set for a multiplication operation $a =$ `(* b c)` uses the same trick: sextuples of relations, constraining not only the pairs $(a,b)$, $(b,c)$ and $(a,c)$ but also the pairs $(a,1)$, $(b,1)$ and $(c,1)$. In this case the trick is even more pertinent since 1 is the multiplicative identity.

**Booleans** The constraint set for the conjunction of booleans is obvious, as shown in Table 10. Boolean disjunction is handled similarly.

| $(b,c)$ | $(c,a)$ | $(b,a)$ |
|---|---|---|
| {ff} | {ff} | {ff} |
| {ft} | {tf} | {ff} |
| {tf} | {ff} | {tf} |
| {tt} | {tt} | {tt} |

Table 10: The constraint set for boolean conjunction: relations when $a =$ `(and b c)`

There are comparison operators that have one boolean output and two inputs of other data types: the `=` and `/=` operators for every data type, and `<`, `<=`, `>=` and `>` on integers. There are only two tuples in these constraint sets. For example, consider the `<` operator on integers: suppose $a =$ `(< x y)`. The tuple of pairs to be constrained is $((x,y),(a,a))$ and there are only two possibilities: $(<, \texttt{tt})$ and $(\geq, \texttt{ff})$. Eliminate one and you have the fullest possible inference; eliminate both and you have a contradiction.

**Bit vectors** The constraint set for the logical-and of bit vectors is straightforward, as shown in Table 11. The logical-or of bit vectors is handled similarly.

**Lists** Two list operators require constraint sets: **append** and **cdr**. The constraint set for $c =$ `(append a b)` contains sextuples of relations: it constrains $(a,b)$, $(b,c)$ and $(a,c)$ and also the relations of each of $a$, $b$ and $c$ with the empty list. The constraint set for $b = \mathrm{cdr}(a)$ constrains triples, constraining the relation $(a,b)$ and also the relations of $a$ and $b$ with the empty list. We could express the cdr relation immediately, as {sc<1, s<1}, but by constraining relations with the empty list at them same time we gain the power to make further inferences. For example, if $a = \mathrm{nil}$ we want to be able to conclude that $b = \mathrm{nil}$.

### 3.4.2 Transitivity constraints

The relations manipulated by the inference engine have transitive properties. When one relation is strengthened, we'd like to take advantage of transitivity to strengthen others, even when the objects in question are not neighbors in the graph.

The inference engine treats transitivity as a kind of triangle property: a restriction on the three binary relations among any three objects. There are $O(n^3)$ of these triangles. To

| $(b,c)$ | $(c,a)$ | $(b,a)$ |
|---|---|---|
| {v00} | {v00} | {v00} |
| {v01} | {v10} | {v00} |
| {v10} | {v00} | {v10} |
| {v11} | {v11} | {v11} |
| {v0x} | {vx0} | {v00} |
| {vx0} | {v00} | {vx0} |
| {v=} | {v=} | {v=} |
| {v=-,v<-} | {vx0} | {vx0} |
| {vx1} | {v1x} | {v=} |
| {v1x} | {v=} | {v1x} |
| {v>-,vxx} | {v>} | {v>} |
| {v>} | {v=} | {v>} |
| {v<} | {v>} | {v=} |

Table 11: The constraint set for bit vector conjunction: relations when $a =$ (and $b$ $c$)

keep the complexity down to $O(n^2)$ the relational constraint algorithm installs transitivity constraints on certain "interesting" triples only, leaving most triples unconstrained. A triple $(a, b, c)$ is deemed interesting if and only if there is a functional constraint on $(a, b)$ or $(b, a)$. Our experiments indicate that this captures many of the useful inferences that occur when full transitivity is used.

Even with full transitivity, it may seem at first glance that an important class of inferences having to do with unary relations is missing. For example, suppose we have established that $x${0+<2+}$y$. This fixes a binary relation between $x$ and $y$, of course, but it also fixes something about each integer independently: $x$ must be 0 and $y$ must be positive. This means that there are restrictions on $(x, z)$ and $(y, z)$ for any $z$ of that same type.

As it happens, this kind of inference occurs as a by-product of inference about transitivity. Given a relation $xRy$, a transitivity constraint will apply to the relations in all triangles with $(x, y)$ on one leg, including the triangle whose other two legs are $(y, x)$ and $(x, x)$. So when any relation restricts $x$'s sign, that will be reflected in the reflexive relation for $(x, x)$. (As mentioned above, this starts out as {--=0, 00=0, ++=0}.) Then, when the relation for $(x, x)$ is strengthened, a transitivity constraint will apply to the relations in all triangles with $(x, x)$ on one leg. The other two legs will be $(x, z)$ and $(z, x)$ for all $z$ of the same type, so the new information about $x$ will be reflected in all relations involving $x$.

### 3.4.3 List property constraints

Sometimes we know that one object is the car or length of another object. This happens with the empty lists, of course; it also happens when the graph contains `length`, `car` or `cons` operators. Whenever we have two lists $a$ and $b$ that both have length objects or both have car objects, we install a list property constraint. This constraint ensures the relation between $a$ and $b$, the relation between car($a$) and car($b$), and the relation between length($a$) and length($b$), are as strong as possible taken jointly. For example, if we have two list objects $a$ and $b$ and two base-type objects $c$ and $d$, and if we know that $c = \text{car}(a)$, $d = \text{car}(b)$, and $a = b$, then the list property constraint for $a$ and $b$ excludes the possibility that $c \neq d$.

The relations of $a$ and $b$ with the empty list contain significant information about their cars and lengths. So a list property constraint $(P, S)$ actually applies to 5-tuples of relations:

$$P = ((a, b), (\mathrm{car}(a), \mathrm{car}(b)), (\mathrm{length}(a), \mathrm{length}(b)), (a, \mathrm{nil}), (b, \mathrm{nil}))$$

### 3.4.4 Transparency constraints

Since the language is purely functional, all functions are referentially transparent: given the same inputs, they produce the same outputs. This applies even to non-terminal functions, about which the inference engine has (or, more accurately, uses) no other information. Given two non-terminal function calls $c_1 =$ `(f` $a_1$ $b_1$`)` and $c_2 =$ `(f` $a_2$ $b_2$`)`, and given that $a_1 = a_2$ and $b_1 = b_2$, we should be able to conclude that $c_1 = c_2$ without knowing anything about the function $f$. Or, given the information that $a_1 = a_2$ and $c_1 \neq c_2$, we should be able to conclude that $b_1 \neq b_2$.

To perform this kind of inference, the inference engine places a constraint on the inputs and outputs for each pair of vertices with the same function. A custom constraint set is built to conform to the number and type of the function's inputs and outputs. (This is the only time Thinner constructs a constraint set—all others are prefabricated.) A pair of $n$-input, $m$-output vertices with the same non-terminal function will get a constraint on $(n + m)$-tuples.

This method could apply to terminal as well as non-terminal vertices: we could construct a custom "transparency" constraint set for each operator, which in addition to transparency could reflect things like associativity (like `or`), bijectivity (like `cons`), or determination by any two of three values (like `*`). The current version of Thinner doesn't do this.

## 3.5 Optimizing constraint computation

Each constraint $(P, S)$ connects a tuple of pairs of objects to a constraint set. The Constrain algorithm uses the constraint set to strengthen the current tuple of relations for those pairs; and, as we'll see, it is often used to compute the same constraint repeatedly. Since it is being called repeatedly with the same constraint set $S$ and with a tuple $Q$ that gets stronger monotonically, it can be optimized to avoid wasted recomputation. The idea is to have each constraint use its own copy of the constraint set $S$, initialized as described previously; then when Constrain finds that the intersection of $Q$ with some tuple $C \in S$ is empty, it discards that tuple from $S$.

**procedure** FastConstrain($Q, S$);
    $Q$ is a tuple of relations *updated in place*;
    $S$ is a set of tuples of relations *updated in place*;
**begin**
    $R :=$ the tuple of bottom relations;
    $T :=$ the empty set;
    **for** each tuple $C$ in $S$
        **begin**
            $I := Q \wedge C$ (the componentwise conjunction);
            **if** $I$ does not contain a bottom relation **then**
                **begin**
                    $R := R \vee I$ (the componentwise disjunction);
                    $T := T \cup \{I\}$

**end**
**end**;
$Q := R$;
$S := T$
**end**;

The initial constraint sets themselves can also be optimized. For example, if a constraint set includes two tuples ($\{$`++=0`$\}, \{$`++<1`$\}$) and ($\{$`++=0`$\}, \{$`++>1`$\}$), it could be reduced by removing those tuples and replacing them with ($\{$`++=0`$\}, \{$`++<1, ++>1`$\}$). This is simple logic minimization, analogous to the optimization of $(a \wedge b) \vee (a \wedge c)$ to $(a \wedge (b \vee c))$. In fact, when the constraint sets used by Thinner are generated, they are, initially, sets of tuples of primitive relations—it's just easier to generate them that way. But before Thinner sees them they are reduced automatically to minimal sets of cubes.

One might take this further and ask: why not dispense with the general constraint set algorithm and instead implement a logically-minimal custom constraint algorithm for each operator? We experimented with this idea by converting a constraint set into logic equations, passing them to a logic minimizer (the Berkeley tool MIS), and converting the resulting logic equations directly into code by hand. The result, unfortunately, was a somewhat slower constraint procedure! Although it was logically minimal it lacked the regular tight-looping structure of the general procedure.

# 4 The relational constraint algorithm

The difficult part of the relational constraint method is the design of suitable vocabularies of relations and the construction of appropriate constraint sets. The relational constraint algorithm itself is quite simple, and the reader has no doubt anticipated it. This section presents the algorithm, gives correctness and complexity results, and offers some tips on implementation.

## 4.1 The algorithm

Here's the algorithm in a nutshell. First, establish a table of relations indexed by pairs of objects and initialize each relation in the table. Next, install the constraints: functional constraints, transitivity constraints, transparency constraints, and list property constraints, as described above. Initialize a set of constraints to be computed; initially, this set contains all the constraints. Now loop while there are more constraints to be computed: choose one, compute it, and update the table of relations; for any entry that got stronger, find the constraints that apply to that pair and add them to the set of those to be computed; and remove the one we just performed.

We need to be able to find all the constraints that apply to a given pair in constant time, so we'll construct a table *Con* indexed by pairs of objects, giving a set of relevant constraints for each pair. The AddCon procedure installs a constraint in the right sets, and also adds the constraint to the set of constraints to be computed.

**procedure** AddCon($(P, S)$, *Con*, *Todo*);
$(P, S)$ is a constraint;
*Con* is a table of sets of constraints indexed by pairs of objects, *updated in place*;
*Todo* is a set of constraints, *updated in place*;

```
begin
    for each pair (i, j) in P
        Con(i, j) := Con(i,j) ∪ (P, S);
    Todo := Todo ∪ {(P, S)}
end;
```

Whenever we strengthen a relation in the table of relations, we also strengthen the reversed relation. (We assume that the new relation is stronger than the old relation.) Then we find all the constraints that apply to either and include them in the set of constraints to be computed.

```
procedure Strengthen(i, j, r, Con, Rel, Todo);
    i and j are objects;
    r is a relation;
    Con is a table of sets of constraints indexed by pairs of objects;
    Rel is a table of relations indexed by pairs of objects, updated in place;
    Todo is a set of constraints, updated in place;
begin
    Rel(i, j) := r;
    Rel(j, i) := r reversed;
    for each constraint (P', S') in Con(i, j) or Con(j, i)
        Todo := Todo ∪ {(P', S')}
end;
```

The installation of all the constraints has one subtlety, already noted: we don't install all the possible transitivity constraints, only those that involve pairs that also have functional constraints. The Init procedure takes a graph $G$ and a set of objects *Obs* and initializes the *Con* table, the table of relations *Rel*, and the set *Todo* of constraints to be computed.

```
procedure Init(G, Obs, Con, Rel, Todo);
    G is a trace graph;
    Obs is the collection of objects for G;
    Con is a table of sets of constraints indexed by pairs of objects, updated in place;
    Rel is a table of relations indexed by pairs of objects, updated in place;
    Todo is a set of constraints, updated in place;
begin
    Todo := the empty set;
    for each pair of identically-typed objects (i, j) from Obs
        begin
            Con(i, j) := the empty set;
            Rel(i, j) := the initial relation for (i, j) (Sections 3.1 and 3.2)
        end;
    for each vertex v in G with a functional constraint
        AddCon({the functional constraint for v}, Con, Todo);
    for each pair of identically-typed objects (i, j) from Obs
        if Con(i, j) is not empty then
            for each object k from Obs of the same type as i and j
                AddCon({the transitivity constraint on (i, j, k)}, Con, Todo);
    for each pair of identically-typed list objects (i, j) from Obs with car or length objects
```

```
    AddCon({the list property constraint for (i,j)},Con,Todo);
for each pair of vertices (v,w) in G with identical non-terminal functions
    AddCon({the transparency constraint for (v,w)},Con,Todo)
    end
end;
```

The input to the main procedure RelationalConstraint is a trace graph $G$; we'll assume that we are also given *Obs*, the set of objects of interest, as described in Section 1.4.

```
procedure RelationalConstraint(G, Obs);
   G is a trace graph;
   Obs is the collection of objects for G;
   Con is a table of sets of constraints indexed by pairs of objects;
   Rel is a table of relations indexed by pairs of objects;
   Todo is a set of constraints;
begin
1  Init(G, Obs, Con, Rel, Todo);
2  while more constraints in Todo do
3      begin
4          (P,S) := a constraint from Todo;
5          Q := the tuple giving the relation Rel(i,j) for each (i,j) in P;
6          Q := Constrain(Q,S);
7          for each pair (i,j) in P and corresponding relation r in Q
8              if r is stronger than Rel(i,j) (that is, if Rel(i,j) ⇏ r) then
9                  Strengthen(i,j,r,Con,Rel,Todo);
10         remove (P,S) from Todo
11     end
end;
```

## 4.2 Correctness

**Theorem 1** *The RelationalConstraint algorithm terminates with a table of relations Rel that cannot be strengthened by the given constraints.*

*Proof:* An invariant for the loop at line 2 in RelationalConstraint is this: if the computation of a constraint would strengthen any relation in *Rel*, that constraint is in *Todo*. This invariant is established by Init, which places all constraints in *Todo*. The invariant is reestablished inside the loop, since whenever any entry in *Rel* is altered, all constraints involving that pair are added to *Todo*; and the one entry removed from *Todo* is the one that was just computed (which would not result in any further strengthening if recomputed).

Let $c$ be the number of constraints installed by Init, and let $k(Rel)$ be the sum over all pairs $(i,j)$ of the distance in the lattice of relations from $Rel(i,j)$ to the bottom relation for that type. Then consider $t = (c+1)\cdot k(Rel) + |Todo|$. On any pass through the loop either some relation is strengthened or no relation is strengthened. If some relation is strengthened then $t$ decreases since $k(Rel)$ is reduced (even though $|Todo|$ may grow as large as $c$). If no relation is strengthened then $t$ decreases since one constraint is removed from *Todo* and none are added. So $t$ always decreases; and when $t = 0$ (if not before) we have $|Todo| = 0$ and the loop at line 2 terminates. Since *Todo* is empty at this point, we can conclude from the invariant that no constraint would strengthen any relation in *Rel* any further. □

This is as much as we can say about correctness: that the algorithm terminates with a table of relations that cannot be further strengthened by the constraints. A more thorough proof of correctness would depend on the correctness and completeness of the constraint sets used.

## 4.3 Complexity analysis

Let $n$ be the number of vertices in the graph. We assume that the trace graph has bounded degree: that is, that functions have bounded input and output arity and that the fanout is bounded. So the number of vertices in the graph, the number of edges in the graph, and the number of objects for the graph, are within constant factors of each other.

**Theorem 2** *RelationalConstraint runs in $O(n^2)$ time.*

*Proof:* First we need to determine how many constraints are installed by Init. The number of functional constraints installed at lines 7-8 is $n$—one for each vertex. At lines 9-12 $O(n^2)$ transitivity constraints are installed: $O(n)$ constraints for each of the $O(n)$ pairs constrained in the previous step. At lines 13-14 $O(n^2)$ list property constraints are installed, at most one for each pair of list objects. Finally, at lines 15-16 $O(n^2)$ transparency constraints are installed (normally far fewer than $n^2$ since the number of pairs of vertices with identical non-terminal functions is likely to be small). The total number of constraints installed by Init is $O(n^2)$, and the time in initialization (line 1 in RelationalConstraint) is also $O(n^2)$.

We now turn to the main loop in RelationalConstraint. The number of times the test at line 8 can be satisfied for a given pair $(i, j)$ is bounded by the height of the lattice of relations, which is constant. So Strengthen is called O(1) times for each pair $(i, j)$. The total time spent in Strengthen is therefore $O(\sum_{(i,j)} |Con(i, j)|) = O(n^2)$. A constraint is computed at most once initially, and then at most once every time one of its pairs is strengthened; this can happen for a given constraint only a constant number of times, so the loop in lines 2-11 is executed $O(n^2)$ times. Disregarding Strengthen, whose contribution has already been counted, each pass through the loop takes constant time. So the total time in the main loop—and the total time in RelationalConstraint—is $O(n^2)$. □

Full computation of transitivity, which is an option left turned off in Thinner, can be achieved by skipping the test at line 10 in Init. This increases the time complexity to $O(n^3)$.

## 4.4 Implementation notes

For simplicity of presentation we have treated *Todo* as a "set"—for asymptotic worst-case analysis it makes no difference which of the waiting constraints from *Todo* is chosen at line 4 in RelationalConstraint. Practically speaking, however, it makes a significant difference. Obviously it's a good idea to take big strengthening steps rather than small ones, and the choice of the next constraint to perform should reflect this:

- We experimented with choosing the next constraint in LIFO, FIFO, and random order. Treating *Todo* as a queue gave the best performance, while treated it as a stack gave the worst. The reason seems to be that "older" constraints is, the more more its pairs may have changed since it was recomputed, and the more likely it is to be effective.
- We experimented with giving preference to transitivity constraints, with giving preference to all other constraints, and with treating all constraints equally. Leaving

transitivity constraints for last gave the best performance, while doing them first gave the worst. The reason seems to be that transitivity constraints tend to be much weaker than, for example, functional constraints, yielding strengthening in smaller steps.

Making the right choices here instead of the wrong ones improved the performance of RelationalConstraint in experiments by about a factor of 3.

The initial *Todo* set contains all constraints; this makes the proof of correctness easier but turns out to be unnecessary. Transitivity, list property, and transparency constraints have no effect on tuples of top relations, which is the initial state of most relations in the table. With a little care these unnecessary initial constraint computations can be avoided.

Lines 9-12 in Init install a number of redundant transitivity constraints. If both $(a, b)$ and $(b, a)$ participate in functional constraints, this algorithm will install transitivity constraints both for the triple $(a, b, c)$ and for the triple $(b, a, c)$, for every $c$. In fact we only need one transitivity constraint per triangle: one for each set $\{a, b, c\}$, not one for each tuple $\{a, b, c\}$. This can be accomplished by ordering the objects, then installing a transitivity constraint for $(a, b, c)$ only if $a \sqsubseteq b \sqsubseteq c$ (and only if $(a, b)$ or $(b, a)$ has a functional constraint).

For simplicity of presentation we have treated *Rel* as one big table indexed by pairs of objects. In fact, we only index using pairs of objects of the same type, and it makes more sense to maintain a separate table of relations for each type.

A more difficult optimization Thinner uses is to reason not about objects but about equivalence classes of objects. Whenever a relation becomes as strong as equality, Thinner merges two equivalence classes and removes a row and column from the table. This makes a big difference in the number of transitivity constraints that are computed, and helps make up for using only $O(n^2)$ of them.

# 5 Examples

In this section we present two simple examples of relational constraint and explain them in detail. The second example is actually not so simple; it demonstrates that in spite of its low complexity, relational constraint sometimes proves surprisingly tricky things.

## 5.1 A simple integer example

Consider the following program:

```
(defun r1 (a b)
    (>= (+ (* a a) b) b))
```

Figure 3 shows the corresponding trace graph. Thinner decides on a set of objects to reason about as described in Section 1.4. There are six integer objects for inference: the input variables $a$ and $b$, the intermediate values labeled $x$ and $y$, and the constants 0 and 1. There are also three boolean objects: the output value labeled $z$, and the constants true and false.

The initial table of relations is shown in Table 12. Here $r_0$ is the initial relation between the constant 0 and any nonconstant integer object and $r_1$ is the initial relation between the constant 1 and any nonconstant. Note that the initial array is symmetric, with relations at least as strong as equality on the diagonal. After relational constraint, the final table of relations is as shown in Table 13; empty slots indicate unchanged relations.

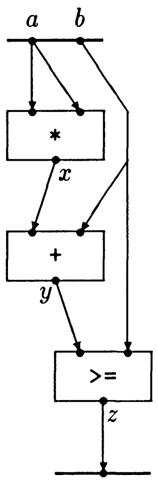


Figure 3: Trace graph for the simple integer example

| | 0 | 1 | $a$ | $b$ | $x$ | $y$ | true | false | z |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 00=0 | 0+<1 | $r_0$ | $r_0$ | $r_0$ | $r_0$ | | | |
| 1 | +0>1 | ++=0 | $r_1$ | $r_1$ | $r_1$ | $r_1$ | | | |
| $a$ | rev($r_0$) | rev($r_1$) | $=$ | $\top$ | $\top$ | $\top$ | | | |
| $b$ | rev($r_0$) | rev($r_1$) | $\top$ | $=$ | $\top$ | $\top$ | | | |
| $x$ | rev($r_0$) | rev($r_1$) | $\top$ | $\top$ | $=$ | $\top$ | | | |
| $y$ | rev($r_0$) | rev($r_1$) | $\top$ | $\top$ | $\top$ | $=$ | | | |
| true | | | | | | | tt | tf | {tf, tt} |
| false | | | | | | | ft | ff | {ff, ft} |
| $z$ | | | | | | | {ft, tt} | {tf, ff} | $=$ |

$$r_0 = \{\texttt{0-<1}, \texttt{0-<2+}, \texttt{00=0}, \texttt{0+<1}, \texttt{0+<2+}\}$$

$$r_1 = \{\texttt{+-<1}, \texttt{+-<2+}, \texttt{+-=0}, \texttt{+0>1}, \texttt{++=0}, \texttt{++<1}, \texttt{++<2+}\}$$

Table 12: Initial relations for the simple integer example

| | 0 | 1 | $a$ | $b$ | $x$ | $y$ | true | false | z |
|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | $r_2$ | | | | |
| 1 | | | | | $r_3$ | | | | |
| $a$ | | | | | $r_4$ | | | | |
| $b$ | | | | | $r_5$ | $\leq$ | | | |
| $x$ | $\mathrm{rev}(r_2)$ | $\mathrm{rev}(r_3)$ | $\mathrm{rev}(r_4)$ | $\mathrm{rev}(r_5)$ | $\{\texttt{00=0}, \texttt{++=0}\}$ | $\mathrm{rev}(r_5)$ | | | |
| $y$ | | | | $\geq$ | $r_5$ | | | | |
| true | | | | | | | | | `tt` |
| false | | | | | | | | | `ft` |
| $z$ | | | | | | | `tt` | `tf` | `tt` |

$$\begin{aligned}
r_2 &= r_0 \setminus \{\texttt{0-<1}, \texttt{0-<2+}\} \\
r_3 &= r_1 \setminus \{\texttt{+-<1}, \texttt{+-<2+}, \texttt{+-=0}, \texttt{++<1}\} \\
r_4 &= \{\texttt{-+<2+}, \texttt{-+=0}, \texttt{++<2+}, \texttt{00=0}, \texttt{++=0}\} \\
r_5 &= r_4 \cup \{\texttt{-0>1}, \texttt{-0>2+}, \texttt{-+>1}, \texttt{-+>2+}, \texttt{+0>2+}, \texttt{+0>1}, \\
&\quad \texttt{++>2+}, \texttt{++>1}, \texttt{-+<1}, \texttt{0+<2+}, \texttt{0+<1}, \texttt{++<1}\}
\end{aligned}$$

Table 13: Final relations for the simple integer example

We've concluded that the output of this function is a constant, the boolean value true. How did this happen? There's no point in reconstructing the actual order of events (since we haven't specified the order in which waiting constraints are performed), but we can analyze the results logically.

The functional constraint for multiplication constrains the relations on pairs $(a, a)$, $(a, x)$, $(a, 1)$ and $(x, 1)$ to make them consistent with $x = a \cdot a$; in particular, this constraint tightens the relation on $(a, x)$ from $\top$ to $r_4$. Note that this subsumes the "law of signs" in that any $x$ satisfying $ar_4x$ must be non-negative. That's not all $ar_4x$ tells us: the magnitude of $x = a \cdot a$ must be either equal to the magnitude of $a$ (this happens when $|a| \leq 1$) or greater than the magnitude of $a$ by at least 2 (this happens when $|a| > 1$). But for our purposes the important thing is that $x$ is non-negative, and after some transitivity constraints this fact is reflected throughout the table, affecting $x$'s entire row and column. This explains $r_2$ and $r_5$: they are the original relations strengthened for $x \geq 0$. ($r_3$ is even stronger: note that it does not allow $x$, the square of an integer, to be 2.)

Now the functional constraint for addition constrains the relations on pairs $(x, b)$, $(x, y)$, $(b, y)$, $(x, 1)$, $(b, 1)$ and $(y, 1)$ to make them consistent with $y = x + b$. Since $x$ is positive, this constraint tightens the relation on $(b, y)$ from $\top$ to $\leq$. This in turn triggers the computation of the functional constraint for the `<=` test, which constrains the relations on $(y, b)$ and $(z, z)$: since $y \geq b$, $z =$ true. Finally, transitivity strengthens the relations between $z$ and the constants true and false.

## 5.2 A tricky list example

Consider the following program:

```
(defun r2 (a b)
    (let ((c (cons 1 a))
          (d (cons 0 nil)))
      (= (append c d) (append b c))))
```

Figure 4 shows the corresponding trace graph. Thinner decides that the output of this program is always the boolean value false. The reader is invited to prove this, using any method, before proceeding.

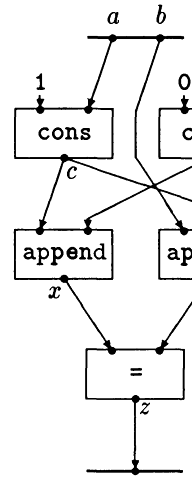


Figure 4: Trace graph for the tricky list example

Perhaps the reader found this inductive proof-by-contradiction. List $x$ is $1 \cdot a \cdot 0$ and list $y$ is $b \cdot 1 \cdot a$. Suppose $x = y$. Then $|b| = 1$, $a$ cannot be nil and its last element must be 0. We also observe that if $a_i = 0$ then $a_{i-1} = 0$ and so, by induction, all the elements of $a$ must be 0. But if $x = y$ the first element of $a$ must be 1. By contradiction, $x \neq y$.

How does Thinner prove this without induction or contradiction? It reasons about seven integer-list objects: the inputs $a$ and $b$, the "let" variables $c$ and $d$, the intermediate values $x$ and $y$, and the empty integer list. There are also three boolean objects: the output $z$ and the boolean constants true and false. The initial table of relations is shown in Table 14. Here $r_0$ is the initial relation between the constant nil and any list, and $r_1$ is the relation established by cons between its input and output lists. Table 15 shows the final table of relations.

The **car** vertices establish initially that 1 is that car of $c$ and 0 is the car of $d$, and that the length of $d$ is the successor of the length of nil (that is, the length of $d$ is 1). Then because the integer relation $0\{$`0+<1`$\}1$ is present (though not shown in the table) the list property constraint for $c$ and $d$ yields $r_4$, which says (among other things) that $c$ and $d$ have different cars.

The functional constraint for the append that computes $x$ constrains the relations on $(c,d)$, $(c,x)$, $(d,x)$, $(c,\text{nil})$, $(d,\text{nil})$ and $(x,\text{nil})$. The constraint set for append reflects the fact that if the length of $d$ is 1 and if $c$ and $d$ have different cars, $c$ cannot be a suffix of `(append c d)`. (This is where that tricky proof is hiding— it's implicit in the constraint set for append.) So when this constraint is computed, we get the relation `c<1` between $c$ and $x$

| | nil | $a$ | $b$ | $c$ | $d$ | $x$ | $y$ | true | false | z |
|---|---|---|---|---|---|---|---|---|---|---|
| nil | $=$ | $r_0$ | $r_0$ | $r_0$ | $r_1$ | $r_0$ | $r_0$ | | | |
| $a$ | rev($r_0$) | $=$ | $\top$ | $r_1$ | $\top$ | $\top$ | $\top$ | | | |
| $b$ | rev($r_0$) | $\top$ | $=$ | $\top$ | $\top$ | $\top$ | $\top$ | | | |
| $c$ | rev($r_0$) | rev($r_1$) | $\top$ | $=$ | $\top$ | $\top$ | $\top$ | | | |
| $d$ | rev($r_1$) | $\top$ | $\top$ | $\top$ | $=$ | $\top$ | $\top$ | | | |
| $x$ | rev($r_0$) | $\top$ | $\top$ | $\top$ | $\top$ | $=$ | $\top$ | | | |
| $y$ | rev($r_0$) | $\top$ | $\top$ | $\top$ | $\top$ | $\top$ | $=$ | | | |
| true | | | | | | | | `tt` | `tf` | {`tf`, `tt`} |
| false | | | | | | | | `ft` | `ff` | {`ff`, `ft`} |
| $z$ | | | | | | | | {`ft`, `tt`} | {`tf`, `ff`} | $=$ |

$$
\begin{aligned}
r_0 &= \{\texttt{sc=0}, \texttt{sc<2+}, \texttt{sc<1}, \texttt{s<2+}, \texttt{s<1}\} \\
r_1 &= \{\texttt{sc<1}, \texttt{s<1}\}
\end{aligned}
$$

Table 14: Initial relations for the tricky list example

| | nil | $a$ | $b$ | $c$ | $d$ | $x$ | $y$ | true | false | z |
|---|---|---|---|---|---|---|---|---|---|---|
| nil | | | | $r_2$ | `sc<1` | `s<2+` | $r_3$ | | | |
| $a$ | | | | $r_1$ | $r_8$ | $r_5$ | $r_6$ | | | |
| $b$ | | | | | $r_8$ | | $r_7$ | | | |
| $c$ | rev($r_2$) | rev($r_1$) | | | $r_4$ | `c<1` | $r_0$ | | | |
| $d$ | `sc>1` | rev($r_8$) | rev($r_8$) | rev($r_4$) | | $r_2$ | $r_9$ | | | |
| $x$ | `s>2+` | rev($r_5$) | | `c>1` | rev($r_2$) | | $r_{10}$ | | | |
| $y$ | rev($r_3$) | rev($r_6$) | rev($r_7$) | rev($r_0$) | rev($r_9$) | rev($r_{10}$) | | | | |
| true | | | | | | | | | | `tf` |
| false | | | | | | | | | | `ff` |
| $z$ | | | | | | | | `ft` | `ff` | `ff` |

$$
\begin{aligned}
r_2 &= r_0 - \{\texttt{sc=0}, \texttt{sc<2+}, \texttt{sc<1}\} \\
r_3 &= r_0 - \{\texttt{sc=0}, \texttt{sc<1}\} \\
r_4 &= \{\texttt{=0}, \texttt{s>2+}, \texttt{s>1}, \texttt{>2+}, \texttt{>1}\} \\
r_5 &= \{\texttt{sc<2+}, \texttt{s<2+}, \texttt{c<2+}, \texttt{<2+}\} \\
r_6 &= r_0 - \{\texttt{sc=0}\} \\
r_7 &= r_6 \cup \{\texttt{c<2+}, \texttt{c<1}\} \\
r_8 &= r_4 \cup \{\texttt{sc=0}, \texttt{sc<1}, \texttt{sc>2+}, \texttt{sc>1}, \texttt{c>2+}, \texttt{c>1}\} \\
r_9 &= r_5 \cup \{\texttt{=0}, \texttt{c<1}, \texttt{<1}, \texttt{s<1}\} \\
r_{10} &= \{\texttt{c>1}, \texttt{c=0}, \texttt{=0}, \texttt{c<2+}, \texttt{c<1}, \texttt{<2+}, \texttt{<1}\}
\end{aligned}
$$

Table 15: Final relations for the tricky list example

shown in the table. It says that $c$ is one shorter that $x$, that they have the same car, but that $c$ is not a suffix of $x$.

On the other hand, the append that computes $y$ constrains its tuple of relations to assert that $c$ is a suffix of $y$ (relation $r_0$). Since $c$ is a suffix of $y$ and not a suffix of $x$, we conclude by transitivity that $x$ and $y$ cannot be equal; and the functional constraint on the equality test settles the value of $z$.

### 5.3 Timings

Thinner is a Common Lisp system running on the Sun Sparcstation. The first inference example, above, is performed by Thinner in 14 milliseconds: 4 milliseconds to initialize the constraints and the table of relations (as in Init) and 10 milliseconds to strengthen the table (as in the main loop of RelationalConstraint). The total number of constraint computations is 38: 33 transitivity constraints and 5 other constraints.

The second inference example, above, is performed by Thinner in 28 milliseconds: 7 milliseconds to initialize the constraints and 21 milliseconds to strengthen the table. The total number of constraint computations is 84: 73 transitivity constraints and 11 other constraints.

## 6 Comparison with other methods

### 6.1 Methods of greater power

One way to reason about programs is to use a more powerful inference system: perhaps a Prolog-style backward-chaining system, or something that finds simple inductive proofs like Boyer and Moore's early system for proving theorems about Lisp programs [BM75]; maybe even a powerful automated reasoning system like Nuprl [C$^+$86].

Our early experiments with a prototype of Thinner used a simple backward-chaining inference engine. Our choice of the vocabulary of relations used by the current Thinner originated in post-mortem examinations of the most useful proofs discovered by that prototype. Looking over these proofs, we made several other observations:

- A trace graph suggests natural paths of inference that a general purpose inference engine ignores. For example, if you've just concluded that $a = 0$, the next natural step is to reason about uses of $a$ in the graph.
- Thinner needs to examine programs and discover any opportunities for thinning—and this means forward, not backward, chaining. What's more, Thinner doesn't need proofs, it only needs conclusions.
- The number of objects in a trace graph is fixed and rather small, and it is these objects about which Thinner needs to reason. There is no real need for unification.

But perhaps it is enough to say that these methods, although powerful, are too costly (that is, slow) for the inference task Thinner presents.

### 6.2 Methods of greater specificity

The literature contains quite a few techniques for dealing with the integer variables of imperative programs. Constant propagation is a time-honored technique, studied and updated in [WZ91]; it treats program variables independently, without considering the relations between them. In [CC76] this kind of analysis is extended to constant intervals.

Relations between integer variables have also been studied: the method of [Kar76] develops a system of affine equations on a program's variables, and the linear restraint analysis of [CH78] develops a system of affine inequalities. A strong motivation in this area has been the compiler-writer's desire to vectorize loops automatically, which requires congruence analysis. In this vein are Granger's congruence analysis, which yields assertions of the form $x \equiv c[m]$ [Gra89] and, later, analysis of congruence equalities, which yields a system of congruence equations [Gra91].

These techniques are too specialized for use in Thinner, which needs to reason about other data types as well. Many are also too expensive: congruence analysis is $O(pn^6 \log^2 n)$ where $p$ is the program size and $n$ the number of integer variables, while linear restraint analysis is exponential.

One interesting aspect of all these techniques is that they have been expressed as *abstract interpretations*, and we now sketch the same treatment for relational constraint.

## 6.3 Abstract interpretation

Abstract interpretation is a framework for the formal treatment of program analysis techniques, first advanced by Cousot and Cousot in [CC77, CC79]; see [AH87] for a modern introduction. Techniques for the analysis of functional programs that have been expressed in this framework include strictness analysis for the optimization of lazy languages [Hug87, Wad87] and update-in-place analysis for the optimization of storage reuse [Hud87]. In this section we sketch a treatment of relational constraint in the framework of abstract interpretation, following [CC79].

**Definition 3** *Given a trace graph $G = (V, E)$, an* execution of $G$ *is a function $e$ that assigns a value to each vertex input and output such that:*

- *For each edge $(v_{out}(i), w_{in}(j))$, $e(w_{in}(j)) = e(v_{out}(i))$. For each edge $(c, w_{in}(j))$, $e(w_{in}(j)) = c$.*
- *For any terminal vertex $v \in V_C$, if $g$ is the function of $v$, $e(v_{out}) = g(e(v_{in}))$.*
- *All inputs to vertices in $V_T$ are assigned the boolean value true.*
- *All inputs to vertices in $V_F$ are assigned the boolean value false.*

Let $E(G)$ denote the set of executions of graph $G$. If $G$ is fully terminal then for each tuple of values $X$, $E(G)$ has at most one execution that assigns $X$ to the inputs of $G$. (This is treated more fully in [Web92], where we also show that for any program and any input tuple $X$, the trace grammar for the program generates at most one fully-terminal graph with an execution on $X$: there is one such graph iff the program terminates on input $X$.) Note that non-terminal vertices in $G$ do not restrict the executions.

An assertion about a given trace graph $G$ is a total function mapping executions of $G$ to {true, false}. Let $\mathcal{A}_G$ be the set of assertions about $G$:

$$\mathcal{A}_G : E(G) \to \{\text{true}, \text{false}\}$$

$\mathcal{A}_G$ ordered by implication forms a complete boolean lattice. We can give a kind of collecting semantics using these assertions. Let $D(G, Q)$ be the set of fully-terminal derivations from graph $G$ in grammar $Q$. For each derivation $d \in D(G,Q)$, $H_d$ denotes the final graph and $m_d$ denotes the function mapping objects in the start graph $G$ to the corresponding objects in the final graph $H_d$. A collecting semantics for a graph $G$ in a trace grammar $Q$ is given by a particular assertion $\mathcal{C}_G$ in $\mathcal{A}_G$:

$$\mathcal{C}_G = \lambda e.[\exists d \in D(G,Q).\exists e_H \in E(H_d).\forall a \in G.e(a) = e_H(m_d(a))]$$

In other words, the only executions of $G$ that satisfy the $\mathcal{C}_G$ are those that match executions of fully-terminal graphs derivable from $G$ in grammar $Q$.

Now for the abstract interpretation: let $A_G$ be the set of tables of expressible relations on pairs of objects in $G$, as defined in Section 2. This is a complete lattice under the partial order $R \sqsubseteq S \equiv \forall(i,j).R(i,j) \sqsubseteq S(i,j)$, using componentwise conjunction and disjunction, the all-$\top$ table as the top element, and the all-$\bot$ table as the bottom element. Elements of $A_G$ represent approximate assertions. The function $\gamma_G : A_G \rightarrow \mathcal{A}_G$, defined as follows, gives meaning to each element of $A_G$.

$$\gamma_G(R) = \lambda e.[\forall(i,j).e(i)R_{(i,j)}e(j)]$$

It should be clear that $\gamma_G$ is not one-to-one: any table $R$ with a bottom element specifies the empty assertion, and any table $R$ that can be strengthened by transitivity specifies the same assertion before and after strengthening. (This is where the transitivity and symmetry computations of relational constraint enter the picture: they find the pairwise-strongest $R' \in A_G$ with $\gamma_G(R') = \gamma_G(R)$.) The range of $\gamma_G$ is a set of assertions $\overline{A}_G \subseteq \mathcal{A}_G$: these are the approximate assertions.

**Theorem 3** *For all assertions $p \in \mathcal{A}_G$ the set $\{q \in \overline{A}_G \mid p \Rightarrow q\}$ has a least element.*

*Proof:* $\overline{A}_G$ contains the supremum of $\mathcal{A}_G$. It is also closed under conjunction: the assertion given by the table $R$ and the assertion given by the table $S$ are both satisfied iff the assertion given by $R \wedge S$ is satisfied—remember that for any two expressible relations their conjunction is also an expressible relation. It follows that $\overline{A}_G$ is a Moore family and any $p \in \mathcal{A}_G$ has a least upper approximation in $\overline{A}_G$. □

We can therefore define an approximation operator $\rho_G : \mathcal{A}_G \rightarrow \overline{A}_G$ as

$$\rho_G(p) = \vee\{q \in \overline{A}_G \mid p \Rightarrow q\}$$

To get a representation for the least upper approximation of an assertion $p$ we need to pick one of the representations $R$ such that $\gamma_G(R) = \rho_G(p)$. We can just choose the conjunction over all such $R$, and so define a function $\alpha_G : \mathcal{A}_G \rightarrow A_G$ as:

$$\alpha_G(a) = \wedge\{R \mid \gamma_G(R) = \rho_G(p)\}$$

Now $\langle \alpha_G, \gamma_G \rangle$ is a pair of adjoined functions in the sense of [CC79].

In [CC79] the operators of the language are expressed as predicate transformers. In our case the operators are predicate constraints: each operator in a graph $G$ yields a constraint $(P,S)$ giving a tuple of pairs $P$ and a description of the space of tuples of relations $S$ that must be satisfied by any table of relations $R \in \overline{A}_G$ consistent with the collecting semantics—that is, any $R$ for which $C_G \Rightarrow \gamma_G(R)$.

Thinner not only approximates $\mathcal{A}_G$ with $A_G$ but also approximates the computation of assertions in $A_G$, treating all non-terminal vertices in $G$ as if they were terminals and insisting only that they be referentially transparent. This is not an inherent limitation of the relational constraint method. We chose this approach for Thinner because the graphs Thinner proposes for semantic analysis are already unfolded: they're interprocedural paths of execution, and the more Thinner explores the longer they get. But it's no more difficult to compute a global table of relations if the application requires it, using a merge-over-all-paths approach.

# 7 Conclusion

The method of relational constraint yields a fairly strong collection of assertions about a program in $O(n^2)$ time. Using a fixed vocabulary of binary relations, a table giving the strongest initial relation known for every pair of objects is constructed. Tuples of relations are then subject to a variety of constraints, which derive from operators in the graph; from transitivity; from functional transparency; or from list properties (which connect list relations to relations on the corresponding car and length objects). Repeated computation of these constraints yields a maximally strong table of relations.

The method should be easy to parallelize. Since the order in which constraints are computed is largely irrelevant, there is no reason why they cannot be computed in parallel. Updates to the table of relations are also unusually order-independent since relations grow stronger monotonically.

Relational constraint need not be the only method of inference used in a given system. Thinner's semantic analyzer also uses the normal forms of expressions to gain information about a program (although this was disabled for the examples discussed in this paper). Relational constraint inference and normal-form comparison seem to complement each other well—each method sees things the other misses. Even a more powerful inference system—say, one with a method for induction—might well seed its initial collection of assertions with those discovered efficiently by the method of relational constraint.

# References


[AH87] Samson Abramsky and Chris Hankin. An introduction to abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1. Ellis Horwood Limited, 1987.

[BH90] H. Bunke and B. Haller. A parser for context free plex grammars. In M. Nagl, editor, *WG '89 Graph-Theoretic Concepts in Computer Science: Proceedings of the 15th International Workshop WG '89*. Springer-Verlag New York Inc., 1990. LNCS 411.

[BM75] Robert S. Boyer and J. Strother Moore. Proving theorems about LISP functions. *Journal of the Association for Computing Machinery*, 22(1), January 1975.

[C+86] Robert L. Constable et al. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, 1979.

[CH78] Patrick Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, 1978.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30, 1989.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91 Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Springer-Verlag, April 1991. LNCS 493.

[Hud87] Paul Hudak. A semantic model of reference counting and its abstraction. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 3. Ellis Horwood Limited, 1987.

[Hug87] John Hughes. Analysing strictness by abstract interpretation of continuations. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 4. Ellis Horwood Limited, 1987.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6, 1976.

[Per89] Mark Perlin. Call-graph caching: Transforming programs into networks. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[Roz87] Grzegorz Rozenberg. An introduction to the NLC way of rewriting graphs. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*. Springer-Verlag, 1987. LNCS 291.

[Wad87] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12. Ellis Horwood Limited, 1987.

[Web] Adam Brooks Webber. Thinning context-free languages. To appear as a Cornell University technical report.

[Web92] Adam Brooks Webber. A formal definition of unnecessary computation in functional programs. Technical Report TR 92-1260, Cornell University, January 1992.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2), April 1991.