Program Derivation by Fixed Point Computation

Jiazhen Cai and Robert Paige[1]

Dept. of Computer Science
NYU/Courant
251 Mercer Street
New York, NY 10012/USA

Abstract

This paper develops a transformational paradigm by which nonnumerical algorithms are treated as fixed point computations derived from very high level problem specifications. We begin by presenting an abstract functional problem specification language SQ^+, which is shown to express any partial recursive function in a fixed point normal form. Next, we give a nondeterministic iterative schema that in the case of finite iteration generalizes the 'chaotic iteration' of Cousot and Cousot for computing fixed points of monotone functions efficiently. New techniques are discussed for recomputing fixed points of distributive functions efficiently. Numerous examples illustrate how these techniques for computing and recomputing fixed points can be incorporated within a transformational programming methodology to facilitate the design and verification of nonnumerical algorithms.

1. Introduction

In a recent survey article [25] Martin Feather has said that the current state of the art of program transformations is still some distance from its ambitious goals - to dramatically improve the construction, reliability, maintenance, and extensibility of software. Our paper represents an attempt towards achieving these goals.

Algorithms often follow the same pattern: find an initial approximation of the solution, and then repeatedly modify the approximation until it becomes the solution. We investigate a class of such algorithms that are all instances of a general nondeterministic iterative algorithm schema for computing least or greatest fixed points of computable functions.

Various fixed point theorems due to Tarski [75], Kleene [45], Cousot and Cousot [18], and others [8] have been applied by Scott to program semantics [69], have been used by Cocke and Schwartz [13], Kildall [44], Tenenbaum [77], and others [36, 42, 16, 67, 43, 74] to specify and implement global program analysis problems, are important to program verification [16, 17, 20, 23], arise in complexity theory [79, 39, 40, 33, 35, 59], and are used to support high level program transformations [2, 29, 7, 49, 62, 53, 56, 11, 54, 70, 60, 81, 51].

We are further encouraged by the following facts:

- Any set generated by inductive definitions can also be defined as the least fixed point of a monotone function [1];
- Without a fixed point operator, a first order language on finite structures cannot express transitive closure [2];
- The language of Relational Calculus [14] over a totally ordered finite domain plus least fixed points of monotone operators precisely expresses all queries computable in polynomial time (in the size of the domain) on a Turing machine [39, 79].

---

[1]Part of this work was done while Paige was a summer faculty at IBM Yorktown and while both authors were at Rutgers University. This work is also partly based upon research supported by the Office of Naval Research under Contract No. N00014-87-K-0461 and by Thomson-CSF/DSE under an ESPRIT contract.

These facts suggest that one reasonable approach to program synthesis is to specify a problem in a least or greatest fixed point normal form, and then apply various subsequent transformations to compile this normal form into an efficiently executable program. Preliminary ideas that support this approach are embodied in a working three-phase prototype compiler that automatically translates abstract problem specifications into efficient RAM code [53, 56]. This compiler, which was implemented by Paige within the RAPTS transformational programming system, has been used to generate many efficient programs of moderate complexity from succinct problem statements, e.g., graph reachability, cycle detection, live code analysis, and attribute closure.

The current version of RAPTS can manipulate problem specifications of the form,

(1) **the** s: $w \subseteq s$ | $s = f(s)$ **minimizing** s

which stands for the smallest set s (with respect to set containment) that contains w and satisfies the equation $s = f(s)$. RAPTS can also manipulate the dual form

the s: $s \subseteq w$ | $s = f(s)$ **maximizing** s

In the first phase of compilation, the system solves these set theoretic equations by transforming them into programs that compute least or greatest fixed points. These transformations introduce a minimal form of algorithmic strategy. In the second phase of compilation, the system uses a generalized finite differencing technique to introduce access paths and basic invariants that serve to implement the strategy efficiently [57, 52, 55]. In the final phase, the low level set-based program that results from the preceding transformations is compiled into conventional code. Elaboration of the three-step automatic programming scheme just sketched is found in [56].

This article makes the following contributions:

- A very high level functional problem specification language SQ^+ is presented. This language contains conventional expressions over boolean and integer datatypes, mathematical dictions found in finite set theory, and least and greatest fixed point expressions. We prove that a subset of SQ^+ with operational semantics can express all partially recursive functions;

- A new nondeterministic algorithm schema for computing least and greatest fixed points of monotone functions is given. This schema generalizes the 'chaotic iteration' found in Kildall [44], Tenenbaum [77], and Cousot and Cousot [17] (restricted to finite iteration), so that it can be adapted in a wider range of contexts to synthesize efficient algorithms and to provide succinct transformational correctness proofs. Based on a lattice theoretic notion of abstract datatype, our fixed point transformations can be applied to a diverse assortment of abstract functions and datatypes;

- Broad sufficient conditions are stated for when specification (1) can be rewritten in the following equivalent way,

  (2) **the** s: $s = w \cup f(s)$ **minimizing** s

  in order to facilitate the mechanical development of regularized iterative procedures with greatly simplified preprocessing operations. The code that results from (2) could be as short as half of that resulting from (1).

- Conditions are given for when the following least fixed point

  the s: $w \subseteq s$ | $s = f(s,t)$ **minimizing** s

  can be recomputed incrementally when either of the parameters w or t increases;

- When f is distributive (i.e., $\forall\ s, t \mid f(s \cup t) = f(s) \cup f(t)$), the following least fixed point

  the s: $w \subseteq s$ | $s = f(s)$ **minimizing** s

  can be recomputed efficiently when w increases or decreases;

- New techniques are presented for solving the lattice theoretic system of equations

$$\textbf{the}\ x_1,\ldots,x_n:\quad x_1 = f_1(x_1,\ldots,x_n),\ \ldots,\ x_n = f_n(x_1,\ldots,x_n)\ \textbf{minimizing}\ x_1,\ldots,x_n$$

efficiently;

- Our use of fixed point theory is shown to uncover new basic principles of software engineering for combining problem specifications and solving them simultaneously using a small number of 'passes'.

Other researchers [7, 62, 49] have employed fixed point transformations applied to general recursion equations. However, their transformations seem less amenable to full mechanization than ours, and they foster a syntactic bias towards a depth-first or breadth-first search implementation.

Before stating our new results, it is worthwhile giving the reader a broader perspective by stepping through the three-phase RAPTS problem specification compiler using a simple case study. (A fuller discussion can be found in [56].)

Example 1: (Graph Reachability) Consider the problem of finding the set of vertices s reachable along paths in a directed graph from an arbitrary set of vertices w. We represent the graph by a finite set of edges e (without multi-edges), where each edge is a pair of vertices. It is convenient to regard e as a multi-valued mapping, so that for each vertex x, the term $e\{x\}$ denotes the set of vertices (called the successor or adjacent vertices of x) reachable from x along a single edge. We also use the image set notation $e[s]$ to mean $\cup_{x \in s} e\{x\}$.

The user can formally define the reachability problem using the following specification:

(3) $\quad \textbf{the}\ s:\ w \subseteq s \mid e[s] \subseteq s\ \textbf{minimizing}\ s$

which represents the smallest set s that contains w and satisfies the predicate $e[s] \subseteq s$. The predicate $w \subseteq s$ appearing in specification (3) signifies that the solution set s includes all paths of length 0. The clause $e[s] \subseteq s$ means that the solution cannot be extended further. The restriction to a minimum solution satisfying these two predicates takes connectivity into account. Without this restriction the entire set of vertices would be a solution.

RAPTS will first transform problem specification (3) into the following equational form:

(4) $\quad \textbf{the}\ s:\ w \subseteq s \mid s = s \cup e[s]\ \textbf{minimizing}\ s$

which stands for the least fixed point of the expression $s \cup e[s]$ in the space of all sets containing w. Next, according to the theory to be discussed, the fixed point p can be computed by executing the following procedure:

```
(5)     p := w                           $Assign w to p.
        (while ∃ x ∈ (e[p] - p) )        $Repeatedly augment p
                 p := p ∪ {x}            $with a vertex adjacent
        end                              $to p.
```

Further improvement in the performance of code (5) can be achieved by applying finite differencing [57, 55] and data structure selection [21, 65, 56]. Finite differencing eliminates a major source of inefficiency within (5) - the repeated calculation of $e[p] - p$ at the top of the **while**-loop. This is achieved by preserving and exploiting the invariant:

$new = e[p] - p$

within the **while**-loop.

Automatic data structure selection will subsequently aggregate the program variables e, p, and new

around the set of graph vertices. That is, we store an array b of records, one record for each node x in the graph. Each record in b contains a node x, a pointer to the set of adjacent nodes $e\{x\}$, and a bit denoting whether x belongs to p or not. For each node x, $e\{x\}$ is represented as an array of pointers to records in b associated with nodes adjacent to x. Variable *new* is represented as a queue of pointers to records belonging to b. The resulting program has a worst case time linear in the number of edges; its worst case auxiliary space is linear in the number of vertices. ■

This paper is structured in the following way. Section 2 presents basic definitions, notational conventions, and a description of the problem specification language SQ^+. Section 3 develops basic transformations to compute fixed points of monotone computable functions. Section 4 describes transformations to compute fixed points dynamically. Section 5 extends earlier transformations to the problem of solving systems of equations. Section 6 sketches an implementation design. The final section surveys related work and discusses open problems.

2. Preliminaries

We first review a few basic definitions and concepts of lattice theory that underlie our main results. This background material may be found in any introductory text on lattice theory; for example, Birkhoff [8] or Gratzer [30]. After that we describe the problem specification language SQ^+ to be used in illustrating transformations for computing and recomputing fixed points.

2.1. Definitions

A *poset* $(L,\leq)$ is a reflexive, transitive, antisymmetric, binary relation $\leq$ on a set L. A poset $(L,\leq)$ has a *minimal* element y iff $\forall\ x \in L \mid (x \leq y \rightarrow x = y.)$ A poset $(L,\leq)$ has a *minimum* element $\mathbf{0}$ iff $\forall\ x \in L \mid x \geq \mathbf{0}$. *Maximal* and *maximum* elements can be defined analogously. A *chain* for a poset $(L,\leq)$ is a strictly increasing or decreasing sequence of elements of L. A poset $(L,\leq)$ is said to have an *ascending* (respectively *descending*) *chain* condition, abbreviated *ACC* (respectively *DCC*), iff there are no infinite increasing (decreasing) chains in L. Let $w \in L$. An element $a \in L$ is w^+-*finite* if the set $\{x \in L \mid w \leq x \leq a\}$ satisfies *ACC*. Similarly, a is w^--*finite* if the set $\{x \in L \mid a \leq x \leq w\}$ satisfies *DCC*. $\mathbf{0}^+$-finite is abbreviated as $\mathbf{0}$-finite, and $\mathbf{1}^-$-finite is abbreviated as $\mathbf{1}$-finite, where we use $\mathbf{1}$ to represent the maximum element in L. Let $a, b, c \in L$. If $a \leq c$ and $b \leq c$, then c is an *upper bound* for a and b. An upper bound c for a and b is said to be the *least upper bound* if every upper bound x for a and b is greater than or equal to c. The least upper bound for a and b is also called the *join* of a and b and is denoted by $a \vee b$. If $a \vee b$ is defined and belongs to L for all $a, b \in L$, then $(L,\leq)$ is called a *join semilattice*. *Lower bounds*, *greatest lower bounds* (also called *meets* and denoted by $\wedge$), and *meet semilattices* are defined analogously. If a poset is both a join and a meet semilattice, then it is a *lattice*. Join and meet operations are each commutative and associative. If T is a set, then the set $P = \{t\text{: } t \subseteq T\}$ of all subsets of T is the *powerset* of T and is denoted by $\mathbf{pow}(T)$. P is a powerset lattice under the relation $\subseteq$ with intersection as meet, union as join, minimum element $\{\}$, and maximum element T.

Let f: $T \rightarrow Q$ be a function from a poset $(T, \leq)$ into a poset $(Q,\leq')$. We say that the *domain* of f is the set $\{x \in T \mid (\exists\ y \in Q \mid f(x) = y)\}$; the *range* of f is the set $\{y \in Q \mid (\exists\ x \in T \mid f(x) = y)\}$. Function f is said to be *partial* if there are elements of T outside its domain; otherwise it is *total*. Function f is said to be *monotone* (respectively *antimonotone*) if for every two elements x, y belonging to its domain such that $x \leq y$, then $f(x) \leq' f(y)$ (respectively, $f(x) \geq' f(y)$). If posets $(T,\leq)$ and $(Q,\leq')$ are the same, then following Gurevich [33], we say that f is *inflationary* (respectively, *deflationary*) at x if $f(x) \geq x$ (resp. $f(x) \leq x$). Function f is said to be inflationary (resp. deflationary) if it is inflationary (resp. deflationary) at each point in its domain. For example, function $x \vee f(x)$ is inflationary for any function f.

Let f: $T \to Q$ be a partial function from a poset (T, ≤) into a poset (Q,≤'). Suppose that we can identify the elements of T and Q with unique finite strings over an alphabet. We say, informally[2], that f is *partially computable* iff there exists a Turing Machine[3] M such that for each element x in T, M terminates with output $y = f(x)$ whenever $f(x)$ is defined and does not terminate otherwise. If f is total and partially computable, then f is *computable*.

2.2. Language

Specification language SQ^+ is essentially a functional subset of the SETL programming language [66] augmented with fixed point operations. In addition to conventional boolean and integer datatypes, SQ^+ includes finite tuples, sets, and maps, which can be nested to arbitrary depth. With a few exceptions to be described, most of the notations in this language are borrowed from finite set theory [72] and conform to universally accepted mathematical notations.

We make use of the overloaded size operator $\#s$ in the following way. If s is a set, then $\#s$ denotes the cardinality of s; if s is a tuple, it denotes the number of components of s. The choice operation $\ni s$ denotes an arbitrary element selected from the set s. If s is empty, then the choice operation has the value Ω, which denotes undefined. We regard a map as a finite set of ordered pairs that maps a domain set to a range set. Thus, a map can be a single-valued function or a multi-valued binary relation. The function retrieval term $f(x)$ denotes the value of function f at domain point x. If x does not belong to the domain of f or if f contains two or more different pairs with first component value x, then $f(x)$ is undefined. We use the image set notation $f\{x\}$ to denote the set $\{y: [u, y] \in f \mid u = x\}$.[4] If s is a set, then the extended image set $f[s]$ denotes the set $\cup_{x\in s} f\{x\}$. We use the tuple selection term $t(i)$ to represent the i^{th} component of tuple t. If t has less than i components, then $t(i)$ is undefined.

If op is a binary, commutative, and associative operation with neutral element e, then it can be applied to all the elements of a set or tuple t using the notation op/t, where

$op/\{\} = op/[\,] = e$

For example, if t is a set of sets, then $\cup/t$ means the same as $\cup_{x\in t} x$. The overloaded binary relational operator ≤ can be used to compare numbers, boolean values (with **false** ≤ **true**), sets (with ≤ representing ⊆), and user-declared partial orderings. Abstract meet (i.e., ∧) and join(i.e., ∨) are used in connection with relation ≤.

We augment SETL with fixed point operations **LFP** and **GFP**. If $f(s)$ is an SQ^+ expression, then the least fixed point operation $\mathbf{LFP}_{\leq,w}(f(s), s)$ denotes the minimum element s (with respect to the partial ordering ≤) that satisfies the condition $w \leq s$ and $s = f(s)$. The greatest fixed point expression $\mathbf{GFP}_{\leq,w}(f(s), s)$ is defined analogously. For convenience, parameter w can be elided when $w = \mathbf{0}$ for least fixed points and when $w = \mathbf{1}$ for greatest fixed points; parameter ≤ can also be elided, whenever the partial ordering is clear within the context. Also, when function f has only one argument, the least and greatest fixed point operators can be abbreviated $\mathbf{LFP}_{\leq,w}(f)$ and $\mathbf{GFP}_{\leq,w}(f)$.

We give a more detailed weakly typed definition of SQ^+ expressions as follows:

1. Every constant (denotations omitted here) and variable is an SQ^+ expression;
2. if

---

[2]See Rogers [64] for a more formal definition of computable functions on sets other than natural numbers.

[3]Turing Machines are defined in Appendix I.

[4]We do not simplify to $\{y: [x, y] \in f\}$, which in SETL means $\{y: \exists\, x \text{ such that } [x, y] \in f\}$.

x is a variable,
f is a binary relation valued SQ^+ expression;
t is a tuple valued SQ^+ expression;
$e, e_1, e_2, \ldots, e_k$ are SQ^+ expressions,
i, i_1, i_2 are integer valued SQ^+ expressions,
$k(x), k_0$ are boolean valued SQ^+ expressions,
$s, s_1, s_2, \ldots, s_k$ are set valued SQ^+ expressions,

then the expressions defined in table 2-1 are also SQ^+ expressions.

Let

$aop \in \{+, -\}$,	$iop \in \{<, \leq, =, >, \geq\}$,
$rop \in \{\subseteq, \subset, =, \supset\}$,	$sop \in \{\cup, -, \cap\}$,
$bop \in$ {**and**, **or**},	$mop \in$ {**min**, **max**},
$op \in$ {$+$, $\cup$, $\cap$, **and**, **or**, **min**, **max**}	$Q \in \{\exists, \forall\}$,

Expression	**Definition**
domain f	$\{\ x : [x,y] \in f\ \}$
range f	$\{\ y : [x,y] \in f\ \}$
f^{-1}	**inverse of** f
$f(e)$	y, **if** $f\{e\} = \{y\}$ Ω **(undefined), otherwise**
$f\{e\}$	$\{\ y : [u,y] \in f \mid u = e\ \}$
$f[s]$	$\{\ y : x \in s,\ y \in f\{x\}\ \}$
$t(i)$	i**th component of tuple** t
$\#s$	**set cardinality**
$\{e(x_1,\ldots,x_k) : x_1 \in s_1, x_2 \in s_2(x_1),$ $\ldots, x_k \in s_k(x_1,\ldots,x_{k-1}) \mid k(x_1,\ldots,x_k)\}$	**set former**
$\{x \in s \mid k(x)\}$	**abbreviation of** $\{x : x \in s \mid k(x)\}$
$[e_1,\ldots,e_k]$	**enumerated tuple**
$\{e_1,\ldots,e_k\}$	**enumerated set**
$e \in s$	**membership test**
$\ni s$	**arbitrary choice**
$i_1\ aop\ i_2$	**arithmetic**
$i_1\ iop\ i_2$	**integer comparison**
$s_1\ sop\ s_2$	**set union, intersection, and difference**
$s_1\ rop\ s_2$	**set comparison**
$k_1\ kop\ k_2$	**boolean and, or**
not k_0	**boolean negation**
$i_1\ mop\ i_2$	**minimum and maximum**
Q $x \in s \mid k(x)$	**boolean valued quantifier (**$\forall$ **or** $\exists$**)**
op/e	$e1\ op\ e2\ op \ldots op\ en$, **where** e **is the set** $\{e1,\ldots,en\}$ **or tuple** $[e1,\ldots,en]$**; evaluates to the neutral element when** $n = 0$
$\mathbf{LFP}_{\leq, w}(e(x), x)$	**least fixed point**
$\mathbf{GFP}_{\leq, w}(e(x), x)$	**greatest fixed point**

Table 2-1: SQ^+ expressions

An SQ expression is an SQ^+ expression with no fixed point operations. An SQ^+ (respectively, SQ) function is a function defined in terms of an SQ^+ (respectively SQ) expression.

In Appendix I, we show that SQ^+ can express all partial recursive functions. Nevertheless, it is convenient to consider expressions syntactically outside of SQ^+ but transformable into SQ^+. Two such examples are the following dual forms of deterministic selection:

(6) **the** s: $w \leq s \mid k(s)$ **minimizing** s

(7) **the** s: $s \leq w \mid k(s)$ **maximizing** s

where $k(s)$ denotes an arbitrary SQ^+ predicate. Specification (6) denotes the minimum element $s \geq w$ with respect to partial ordering $\leq$ such that predicate $k(s)$ holds. Specification (7) denotes the maximum element $s \leq w$ for which $k(s)$ holds. In either case, if there is no unique solution, then the expression value is undefined. These expressions are transformable into SQ^+ whenever $k(s)$ can be turned into the form $s = f(s)$, where $f(s)$ is an SQ^+ function.

In dealing with systems of equations it is sometimes convenient to use the following notation

$$
\begin{array}{ll}
\text{(i)} & \textbf{the } x_1,\ldots,x_n\text{: } w_1 \leq_1 x_1,\ldots,w_n \leq_n x_n \\
& \quad \mid\ x_1 = f_1(x_1,\ldots,x_n) \\
& \qquad \ldots \\
& \qquad x_n = f_n(x_1,\ldots,x_n) \\
& \textbf{minimizing } x_1,\ldots,x_n
\end{array}
$$

to represent the SQ^+ expression, $\mathbf{LFP}_{\leq,w}([f_1(x), \ldots, f_n(x)], x)$, where $f_i(x)$, $i = 1,\ldots,n$, are SQ^+ functions and $\leq$ is component-wise comparison (using $\leq_i$ for the ith components, i=1,..,n) of n-tuples. We also use the notation

$$
\begin{array}{ll}
\text{(ii)} & \textbf{the } x_1,\ldots,x_n\text{: } x_1 \leq_1 w_1,\ldots,x_n \leq_n w_n \\
& \quad \mid\ x_1 = f_1(x_1,\ldots,x_n) \\
& \qquad \ldots \\
& \qquad x_n = f_n(x_1,\ldots,x_n) \\
& \textbf{maximizing } x_1,\ldots,x_n
\end{array}
$$

to stand for $\mathbf{GFP}_{\leq,w}([f_1(x), \ldots, f_n(x)], x)$. If n-tuple $[z_1,\ldots,z_n]$ is the value of expression (i), then any n-tuple $[y_1,\ldots,y_n]$ that satisfies the n inequalities and the n equalities in (i) must also satisfy $z_j \leq_j y_j$, $j = 1,\ldots,n$. Expression (ii) is defined analogously. If there is no unique minimum (respectively maximum) n-tuple satisfying the inequalities and equalities in (i) (respectively (ii)), then the expression is undefined.

We will sometimes employ several convenient abbreviations for SQ^+ expressions. Multi-variate function application $f([x_1,\ldots, x_n])$ is abbreviated $f(x_1,\ldots,x_n)$, and multi-variate multi-valued map application $f\{[x_1,\ldots,x_n]\}$ is abbreviated $f\{x_1,\ldots,x_n\}$. It is useful to abbreviate set operations $s \cup \{x\}$ and $s - \{x\}$ by s **with** x and s **less** x, respectively. It is sometimes useful to define and apply nonrecursive functions (without procedure parameters) with call-by-name semantics. Thus, if $e(x)$ is an SQ^+ expression that depends on variable x, we can define function $f(x) = e(x)$, and use f as if it were a finite map; e.g., if s is a set, then we can use image set notation $f[s]$ to abbreviate the set $\{e(x)\colon x \in s\}$. The following kind of set former

$$\{x \in s \mid k(x) \ \textbf{minimizing}\ f(x)\}$$

can be used to abbreviate the more cumbersome SQ^+ expression

$$\{x \in s \mid k(x) \ \textbf{and}\ f(x) = \textbf{min}/\{f(y)\colon y \in s \mid k(y)\}\}$$

We can attempt to provide SQ^+ with an operational semantics using a lower level imperative language containing assignment statements, conditional statements, **while**-loops, and other control structures. Assignment statements of the form $x := x\ op\ y$ can be abbreviated $x\ op{:=}$ y. Hence, set element addition is denoted by s **with**:= x, and element deletion is denoted by s **less**:= x. We use the basic control structure

$$
\begin{array}{l}
(\textbf{for } x \in s) \\
\quad block(x) \\
\textbf{end}
\end{array}
$$

to execute *block* for all values of x belonging to s. If s is a set, then we execute *block* for each value of x without repetition and in any order. If s is a tuple, then *block* is executed for every component value of s from the first to the last component.

3. Fixed Point Computation

This section summarizes and generalizes the basic fixed point transformations used in the RAPTS system and the fixed point theory underlying them. The purpose of these transformations is to turn functions expressed in fixed point normal forms into executable programs so that their efficiency can be further improved by finite differencing. In the contexts to which they apply, the fixed point transformations to be discussed provide a formal basis for problem solving by iteration. Illustration of these transformations in numerous examples suggests a wide range of application. Although lacking a precise characterization of what functions can be transformed into efficient computations by these techniques, we can, at least, prove that all partially computable functions can be solved as fixed point computations. The proof is given in Appendix I by expressing a Turing Machine in SQ^+.

Although the theory and transformations to be described in this section allow us to compute fixed points within general posets and semilattices, it is convenient and useful to illustrate many applications with collections of sets. This is because

1. the set is one of the simplest and most commonly used aggregate data objects;
2. the compact but powerful operations of set theory can be used naturally to express combinatorial algorithms succinctly;
3. basic set operations frequently satisfy the conditions of these transformations (e.g. monotonicity);
4. it is often possible to recognize the transformational conditions automatically; this will be discussed later.

For succinctness we will develop our theory and give transformations for least fixed points. To obtain dual forms of theorems, definitions, conditions, and transformations so that they apply to greatest fixed points, we need to switch **LFP** and **GFP**, $>$ and $<$, $\leq$ and $\geq$, **0** and **1**, $\vee$ and $\wedge$, *ACC* and *DCC*, inflationary and deflationary, **minimizing** and **maximizing**, and w^+-finite and w^--finite. Explicit transformations for greatest fixed points can be found in [9].

3.1. Basic Theory

All of our fixed point transformations are derived from the following theorem and corollary, which can be derived from Tarski's more general Theorem [75] or its constructive reformulation due to Cousot and Cousot [18]:

Theorem 1: (Paige and Henglein [56]) Let $(T,\leq)$ be a poset with a unique minimum element designated **0**. Let f: $T \rightarrow T$ be a monotone computable function. Then the set $\{f^i(\mathbf{0})$: $i = 0, 1,... \}$ is finite iff there exists an integer $k \geq 0$ such that $f^k(\mathbf{0}) = \mathbf{LFP}(f)$.

In the reachability problem (cf. Example 1), what we really need is not the least fixed point, but the least fixed point that contains the source set w. We call this a *conditional least fixed point*. In general, we use the term $\mathbf{LFP}_{\leq,w}(f(s), s)$ to denote the conditional least fixed point of f that is greater than or equal to w. The following Corollary extends Theorem 1 to conditional least fixed points.

Corollary 2: Let $(T,\leq)$ be a poset . Let f: $T \rightarrow T$ be a monotone computable function, $w \in T$, and $w \leq f(w)$. Then the set $\{f^i(w)$: $i = 0, 1,... \}$ is finite iff $\mathbf{LFP}_{\leq,w}(f) = f^k(w)$ for some integer $k \geq 0$.

Proof: The subspace $T' = \{x \in T \mid x \geq w\}$ together with relation $\leq$ form a poset with minimum element w. Since f maps T' into itself, Theorem 1 applies. ■

Since any function f, as defined in Corollary 2, must be inflationary at **0**, Corollary 2 is a generalization of Theorem

1. The inflationary condition $w \le f(w)$ in Corollary 2 is important. Without that, the conclusion may be incorrect, as is shown by the following example:

Example 2: Consider a relation $f = \{[a, a], [a, b], [b, c], [c, c]\}$. Let $w = \{b, c\}$. Then $\mathbf{LFP}_{\subseteq,w}(f[s], s)$ must be $s = \{a, b, c\}$. But $f^k(w) = \{c\}$ for any $k \ge 1$. ■

In order to design and implement program transformations based on Corollary 2, it is useful to consider various properties that imply the finiteness of the set $\{f^i(w)\text{: } i=0,1,...\}$.

Theorem 3: Let $(T,\le)$ be a poset. Let $f\text{: } T \to T$ be a monotone computable function, $w \in T$, and $w \le f(w)$. Then the set $\{f^i(w)\text{: } i = 0,1,...\}$ is finite if any one of the following conditions holds:

1. either of the sets $\{x \in T \mid w \le x\}$ or $\{x \in$ **range** $f \mid w \le x\}$ is finite;
2. either of the sets $\{x \in T \mid w \le x\}$ or $\{x \in$ **range** $f \mid w \le x\}$ satisfies *ACC*;
3. f has a w^+-finite fixed point greater than or equal to w either with respect to T or to the poset (**range** $f, \le$);
4. The poset $(T, \le)$ is a join semilattice and function f has the form $f(x) = x \vee g(x)$, where the set $\{g(x)\text{: } x \in T \mid w \le x\}$ is finite.
5. The poset $(T, \le)$ is a join semilattice, function f has the form $f(x) = x \vee g(x)$, g is monotone, and the set $\{g(x)\text{: } x \in T \mid w \le x\}$ satisfies *ACC*.

Proof: 1, 2 and 3 are simple. 4 is true because there can be only a finite number of new points that result from taking joins of points in the range of g; i.e., the set $\{f^i(w)\text{: } i = 1, 2,... \} = \{ w \vee g(w) \vee ... \vee g(f^{i-1}(w))\text{: } i = 1,2,...\}$ is finite. For condition 5 monotonicity of g implies that $f^i(w) = w \vee g(f^{i-1}(w))$, $i = 1,2,...$ Also by monotonicity of g and *ACC* property, the sequence $g(f^{i-1}(w))$, $i = 1,2,...$, can only form a finite ascending chain. Hence, the set $\{w \vee g(f^{i-1}(w))\text{: } i = 1,2,...\}$ must be finite. ■

It is interesting to note that the set $\{f^i(w)\text{: } i=0,1,...\}$ can be finite while **LFP**(f) is not w^+-finite. It might be the case that there is an infinite ascending chain between $f^i(w)$ and $f^{i+1}(w)$ for some $i \ge 0$.

Example 3: (Cycle detection) A finite directed graph e contains a cycle iff the largest subset s of vertices each containing a successor belonging to s is nonempty. A more formal specification of cycle detection is,

(8) (**the** s: $s \subseteq$ **domain**$(e) \cup$ **range**(e) |
 $(\forall x \in s \mid e\{x\} \cap s \neq \{\})$ **maximizing** s) $\neq \{\}$

Since specification (8) is equivalent to the test of whether the greatest fixed point of the monotone expression

$$s - \{x \in s \mid e\{x\} \cap s = \{\}\}$$

(that is also a subset of **domain**$(e) \cup$ **range**(e)) is nonempty, we can compute this fixed point efficiently according to the dual forms of condition 1 of Theorem 3 and Theorem 1. ■

Example 4: Consider the problem of graph reachability again. The set of vertices p that are reachable from w along edges in e can be found by initializing p to w, and storing successive values of $p \cup e[p]$ into p until $p = p \cup e[p]$. Since the range of e is finite, condition 4 of Theorem 3 guarantees that after a finite number of steps, p will be the least fixed point of $s \cup e[s]$ in the space of all sets that include w. ■

Example 5: Kildall's form of the constant propagation problem [44] satisfies the dual form of condition 2 of Theorem 3. For this example T is an uncountably infinite space of finite functions $h\text{: } N \to (F \cup \{\mathbf{I}\})$, where N is a finite set of nodes in a directed graph that models control flow, F is the set of partially defined finite functions $f\text{: } V \to \mathbf{R}$, V is a set of program variables, $\mathbf{R}$ represents the set of real numbers, and $\mathbf{I}$ is a special maximum element of F (which is ordered by set containment) with the property that $\forall\, g \in F$, g

$\subseteq$ **I**. If T is ordered by node-wise set containment (i.e, $\forall\, h_1, h_2 \in T$, $h_1 \le h_2$ iff $\forall\, n \in N$, $h_1(n) \subseteq h_2(n)$), then we see that a *DCC* holds and that the length of the longest chain in T is $1 + (\#N)(\#V)$.[5] ■

Many of the algorithms that perform global program optimization were designed and proved correct by first formulating them as fixed point computations. In the early development of this field Cocke and Schwartz [13] and Kildall [44] justified convergence of their algorithms using condition 1 of Theorem 3 and restricted function f to be *distributive*; i.e., for every $x,y \in T$, $f(x \vee y) = f(x) \vee f(y)$. (Note that distributivity implies monotonicity, but monotonicity does not imply distributivity.) Tenenbaum [77] and Kam and Ullman [42, 41] later designed algorithms based on the more general condition 2 of Theorem 3 and the more general monotonicity property for function f.

3.2. Basic Fixed Point Transformations

According to Corollary 2 a straightforward algorithm to compute $\mathbf{LFP}_{\le,w}(f)$ initializes p to w, and then repeatedly computes a new value of p by assigning $f(p)$ to p until p does not change. The final value of p is the solution. Although such an iterative procedure may be efficient, it may also be highly inefficient in our set theoretic applications, because of the potentially costly redundancy in the recomputation of $f(p)$ each iteration. For example, when applied to the reachability problem (cf. Example 1), the repeated recomputation

(9) $\quad p := p \cup e[p]$

is unsatisfactory because the new approximation of p is completely recomputed and copied each iteration, even though it may differ only slightly from its old value. Another shortcoming with the iterative step (9) is that it is biased towards a breadth-first search strategy.

The following Theorem illustrates two forms of nondeterministic iteration that can overcome both problems.

Theorem 4: Let $(T,\le)$ be a poset. Let $f\colon T \to T$ be a monotone computable function, $w \in T$, and $w \le f(w)$. Let $s_0,\ldots,s_i,\ldots$ be any sequence such that

1. $s_0 = w$;
2. $s_{i+1} \in \{x \in T \mid s_i \le x \le f(s_i)\}$, $i = 0,1,\ldots$ (Note that such sequences always exist for computable functions f that are monotone and inflationary at w.)

Then we conclude the following.

(a) If there exists an integer $k \ge 0$ such that $s_k = f(s_k)$, then $s_k = \mathbf{LFP}_{\le,w}(f)$.

(b) If $\mathbf{LFP}_{\le,w}(f)$ is w^+-finite, and if $s_i < s_{i+1}$ whenever $s_i \ne f(s_i)$, then there exists an integer $k \ge 0$ such that $s_k = f(s_k)$.

Proof: (a) We use a simple dominated convergence argument. By assumption, s_k is a fixed point of f that is greater than or equal to w. Let $p \ge w$ be any other fixed point of f. Then by condition 2 and properties of f,

$$w \le s_i \le f^i(w) \le p,\ i = 0,1,\ldots$$

Hence, $s_k = \mathbf{LFP}_{\le,w}(f)$. Part (b) follows immediately from the proof of part (a) and the definition of w^+-finite. ■

Conditions (a) and (b) of Theorem 4 imply two forms of iteration. In the most general form satisfying

[5] Of course, any algorithm that attempts to solve this problem would only approximate real numbers by using finite representations (as can be done with rationals).

condition (a), we have a nondecreasing sequence s_i, i=0,1,..., that converges to a fixed point of f after a finite number of steps. (Note that the sequence does not have to be strictly increasing prior to convergence.) Thus, in order to ensure that a sequence satisfies condition (a), we must prove directly that such convergence occurs. We refer to this as the 'operational' approach.

Based on iteration according to condition (b) we can develop a robust, and perhaps easier, 'algebraic' approach to fixed point computation. In this approach sequences are strictly increasing and must converge because of w^+-finiteness.

We formalize the way in which sequences are generated in the algebraic approach as follows. Let $(T, \leq)$ be a poset, and let S be a nonempty set. A partially defined function Δ: $T \times T \rightarrow \mathbf{pow}(S)$ is called a *workset* function if $\Delta(q, p) = \{\} \Leftrightarrow q \leq p$ for all $[q,p] \in$ **domain** Δ. A partially defined function δ: $T \times S \rightarrow T$ is called an *increment* function if $\delta(p,z) \geq p$ for all $[p,z] \in$ **domain** δ. The two functions Δ and δ are said to be *feasible*[6] relative to a partial function f: $T \rightarrow T$ at a point $w \in T$ if the following conditions hold:

1. $s \vee f(s)$, $\Delta(f(s), s)$, and $\delta(s, z)$ $\forall z \in \Delta(f(s), s)$ are defined;
2. $\forall z \in \Delta(f(s), s) \mid s < \delta(s, z) \leq s \vee f(s)$;

for all s belonging to every sequence,

$$
\begin{aligned}
(10)\quad & s_0 = w \\
& s_{i+1} = s_i \text{ if } \Delta(f(s_i), s_i) = \{\} \text{ and} \\
& s_{i+1} \in \{\delta(s_i, r) : r \in \Delta(f(s_i), s_i)\} \text{ otherwise, } i = 0, 1, \ldots
\end{aligned}
$$

If Δ and δ are feasible relative to function f at point w, then any sequence (10) is said to be *generated* by Δ and δ at w.

Transformation 1: Let f: $T \rightarrow T$ be a monotone computable function, where $(T, \leq)$ is a poset. Let $w \in T$ and $w \leq f(w)$. If $\mathbf{LFP}_{\leq,w}(f)$ is w^+-finite and Δ and δ are feasible relative to f at w, then the following transformation is correct:

```
        p := LFP≤,w(f)
          ⇒
(11)    p := w
        (while ∃ z ∈ Δ(f(p), p))
              p := δ(p, z)
        end
```

Proof: The successive values assigned to p in program (11) form a sequence s_i, i=0,..., generated by functions Δ and δ, which are feasible relative to f at w. By definition of feasibility, such a sequence is strictly increasing as long as $f(s_i) \neq s_i$; also $s_{i+1} \leq f(s_i)$, i=0,... Since $\mathbf{LFP}_{\leq,w}(f)$ is w^+-finite, by Theorem 4 (b) there exists a finite integer $j \geq 0$ such that $s_j = \mathbf{LFP}_{\leq,w}(f)$. Hence, $f(s_j) = s_j$, which implies that $\Delta(f(s_j), s_j) = \{\}$, since Δ is a workset function. Consequently, code (11) halts after j iterations of the **while** loop. ■

For a poset $(T, \leq)$ and monotone computable function f: $T \rightarrow T$ that is inflationary at $w \in T$, we can always choose $S = T$ and the following functions feasible relative to f at w:

$$
\begin{aligned}
(12)\quad & \text{for all } p, q \in T, \text{ where } q \vee p \text{ is defined,} \\
& \Delta(q, p) = \{\} \text{ if } q \leq p, \text{ and } \{q\} \text{ otherwise;} \\
& \delta(p, q) = p \vee q;
\end{aligned}
$$

Using functions (12) Transformation 1 leads to the conventional iteration implied by Corollary 2; that is,

---

[6]In its dual formulation for greatest fixed points the feasible functions include a workset function Δ that must satisfy the condition $\Delta(q, p) = \{\} \Leftrightarrow q \geq p$ for all $[q,p] \in$ **domain** Δ. However, instead of an increment function we use a *decrement* function δ that must satisfy the condition $\delta(p,z) \leq p$ for all $[p,z] \in$ **domain** δ.

```
(13)  p := w
      (while f(p) >  p)
        p := f(p)
      end
```

If we redefine Δ,

$$\Delta(q,\ p) = \{z\colon p < z \le q \vee p\} \text{ if } q \vee p \text{ is defined}$$

then we obtain precisely the nondeterministic iteration given in Theorem 4 (b).

Transformation 1 turns functional programs involving only input and output variables into imperative ones by introducing an assignment statement and the intermediate variable p. In the case of (11), the value of p starts out at w and each transition from an old value of p, p_{old}, to a new value of p, p_{new}, is governed by the invariant $p_{new} \in h(p_{old})$, where $h(p) = \{\delta(p, z)\colon z \in \Delta(f(p), p)\}$. Since the value of p_{new} can be chosen from a set of values, various transformations can be tailored from (11) to make this choice of values based on highly efficient strategies.

One guiding principle in designing efficient strategies with Transformation 1 is to avoid the potentially costly computation $\Delta(f(p), p)$. This can sometimes be achieved using finite differencing to preserve a program invariant that keeps the value of $\Delta(f(p), p)$ stored at the point where it is needed within the **while**-loop predicate. This approach is profitable whenever the cumulative cost of preserving such an invariant is asymptotically lower than the cost of computing $\Delta(f(p), p)$ each time through the loop. Of course, it is also useful to keep the size of the set $\Delta(f(p), p)$ down to conserve space. Another more vaguely stated principle is to generate the next element $\delta(p, z)$ in a way that makes progress at minimal cost. This can sometimes be achieved by choosing the next element p_{new} by augmenting the current element p_{old} with a minimal increment dp such that $p_{new} = \delta(p_{old},dp) > p_{old}$. Of course, we should also exploit any local simplifications or optimizations to implement assignment $p := \delta(p,dp)$ as efficiently as possible.

Below we illustrate Transformation 1 with a few computable problems discussed in [3]:

Example 6: (Reachability continued) If we define $f(s) = e[s] \cup s$, then f must be inflationary at w. It is also easy to see that workset function $\Delta(f(s), s) = f(s) - s$ and increment function $\delta(s, z) = s$ **with** z are feasible relative to f at any vertex set w. Hence, by Transformation 1 we can compute the set of vertices p that are reachable from w along edges in e by initializing p to w and repeatedly augmenting p with a single arbitrary element selected from $e[p] - p$, until $e[p] - p$ is empty. Such a nondeterministic reachability algorithm can subsequently be refined into a variety of strategies with efficient implementations. ■

Example 7: (The single source shortest path problem) Consider a graph $G = (V, E)$, where V is a set of vertices and E is a set of ordered pairs representing edges. Given a source node $s \in V$ and a weight function c that maps E to nonnegative reals $\mathbf{R}^+$ extended with a maximum element ∞, we want to compute the function $d = \{[i, a]\colon i \in V, a$ is the length of the shortest path from node s to node $i\}$. If we consider the poset of all functions $m\colon V \to \mathbf{R}^+$ under the ordering $m_1 \le m_2$ *iff* $(\forall i \in V \mid m_1(i) \le m_2(i))$, then the solution d is the greatest fixed point of the function $f(m) = \{[i, a]\colon i \in V, a = \mathbf{min}/\{m(i), \mathbf{min}/\{m(j) + c(j, i)\colon j \in E^{-1}\{i\}\}\}\}$ that is less than or equal to $\{[s, 0]\} \cup \{[i, \infty]\colon i \in V \mid i \ne s\}$. With suitable feasible functions Δ and δ we could apply the dual form of Transformation 1 to obtain the following naive algorithm to compute d:

```
d := {[s, 0]} ∪ {[i, ∞]: i  ∈ V | i ≠ s}
(while ∃ [k,a] ∈ {[i,q]: i ∈ V, q = min/{d(j)+c(j,i): j ∈ E⁻¹{i}}
                                     | d(i) > q})
  d(k) := a
end
```

The preceding code solves the problem but is too inefficient. Even after finite differencing is applied to avoid the cost of computing Δ, this algorithm would converge too slowly. However, if we choose

$$\Delta(f(d),d) = \{[i,a]:\ i \in V,\ a = \mathbf{min}/\{d(j) + c(j,i):\ j \in E^{-1}\{i\}\}\ |\ d(i) > a\ \mathbf{minimizing}\ a\},$$

then it will converge within n iterations. ■

The next two examples illustrate the two different approaches suggested by Theorem 4 to compute the same fixed point. The first example illustrates the algebraic approach, and the second example illustrates the operational approach.

Example 8: (Single function coarsest partition problem) If s is a finite set, then a *partition* P of s is a set of pairwise disjoint subsets of s whose union is all of s. The elements of P are called its *blocks*. If P and Q are two partitions of s, then Q is a *refinement* of P (denoted by $Q \le P$) if every block of Q is contained in a block of P. Observe that the space of partitions over s forms a lattice with maximum element $\{s\}$ and minimum element $\{\{x\}: x \in s\}$.

The Single Function Coarsest Partition Problem inputs a finite set s, an initial partition P of s, and a total function h on s. It outputs the coarsest (i.e., maximal) refinement Q of P such that $\forall\ b \in Q\ \exists\ d \in Q\ |\ h[b] \subseteq d$. In [54] this problem is reformulated as computing the greatest fixed point (that is also a refinement of P) of the following monotone function

$$f(Q) = \{b \cap h^{-1}[q]:\ b \in Q,\ q \in Q\ |\ b \cap h^{-1}[q] \ne \{\}\}$$

which maps a partition Q into a refinement of Q.

The first step in a derivation of an efficient algorithm is to apply the dual form of Transformation 1 with the following feasible functions relative to f at P (with a feasibility proof left to the reader):

$$\Delta(f(Q),Q) = Q - f(Q)$$ [7]

and

$$\delta(Q,q) = (Q - \{q\}) \cup \{q - b, b\}$$
$$\mathbf{where}\ b \in \{x \in f(Q)\ |\ x \subset q\ \mathbf{and}\ \#x \le \#q/2\}$$ [8]

The preceding feasible functions combine Hopcroft's 'choose the smaller half' strategy with the double partition approach that Paige and Tarjan used to solve the relational coarsest partition problem [58]. Application of finite differencing to preserve the values of $f(Q)$ and Q - $f(Q)$ incrementally leads to an algorithm with the same $O(n\mathbf{log}n)$ time bound as Hopcroft's [38]. ■

It is interesting to consider an alternative operational derivation of an algorithm to solve the single function coarsest partition problem based on Theorem 4 (a).

Example 9: According to Example 8, we want to compute $\mathbf{GFP}_{\le,P}(f)$. Consider the following function,

$$split(P,b) = \{t \cap h^{-1}[b]:\ t \in P\ |\ t \cap h^{-1}[b] \ne \{\}\ \} \cup \{t - h^{-1}[b]:\ t \in P\ |\ t - h^{-1}[b] \ne \{\}\ \}$$

which is easier to compute than $f(P)$. Since $P \ge split(P,b) \ge f(P)$, $\forall\ b \in P$, then any sequence of partitions, $P_0 = P$ and $P_{i+1} = split(P,b)$ where $b \in P_i$, i=0,1,..., is nonincreasing and satisfies the general conditions of Theorem 4.

Such a sequence of partitions P is generated by an initial partition and the following nondeterministic code:

---

[7] $\Delta(f(Q),Q)$ stores those blocks of Q not belonging to $f(Q)$.

[8] Note that $\delta(Q,q)$ is nondeterministic in the sense that block b is any block of refinement $f(Q)$ that has no more than half the elements of block q of partition Q, where $b \subset q$.

```
(14)      Q := {s}
         (while ∃ q ∈ (Q - P), b ∈ P | b ⊂ q and #b ≤ #q/2)
           P := split(P,b)
           Q := (Q - {q}) ∪ {q - b, b}
         end
```

Code (14) preserves the invariants $Q \geq P$ and $\forall t \in Q \mid P = split(P,t)$. This code terminates with $Q = P$, because the sequence of Q partitions generated forms a descending chain in a finite lattice. Since $split(P,b) = P$, $\forall\, b \in P$, iff P is a fixed point of f, then by Theorem 4 (b) and (a) the final value of P is $\mathbf{GFP}_{\leq,P}(f)$. ■

It is interesting to note that application of the *split* function in code (14) preserves the invariant $P = f(Q) \wedge P_0$, where P_0 is the initial value of P. Also, the modification to Q occurring within (14) is an efficient implementation of $\delta(Q,q)$ in Example 8. Hence, the implementations that would be derived in Examples 8 and 9 are much the same.

In the next two subsections we tailor efficient variants of fixed point Transformation 1 for special kinds of functions and posets.

3.3. Special Functions

It is useful to refine Transformation 1 to compute fixed points for several particular kinds of functions $f(s)$. These include the 'inductive' form $s \vee g(s)$ and its generalization $\vee_{i=1}^{k} g_i(s)$, the tupling form $[f_1(s),...,f_n(s)]$, parameterized forms $g(s,...,s)$ with m occurrences of variable s within function g, composition $h \circ g$, and the fixed point form $\mathbf{LFP}_{\geq,w}(g(s,t),t)$.

Let us first consider a special case of Transformation 1 when $f(s) = s \vee g(s)$.

Transformation 2: (Inductive form)

Let f: $T \to T$ be a monotone computable function $f(s) = s \vee g(s)$, where $(T, \leq, \vee)$ is a join semilattice with a minimum element $\mathbf{0}$ and $w \in T$. If $\mathbf{LFP}_{\leq,w}(f)$ is w^+-finite and functions Δ and δ are feasible relative to g at w, then the following transformation is correct:

```
          p:= LFP≤,w(s ∨ g(s),s)
⇒
(15)      p := w
          (while ∃ z ∈ Δ(g(p), p))
              p := δ(p,z)
          end
```

Proof: The successive values assigned to p in program (15) form a sequence s_i, i=0,..., generated by functions Δ and δ, which are feasible relative to g at w. By definition of feasibility such a sequence is strictly increasing as long as $g(s_i) \not\leq s_i$. But $g(s_i) \not\leq s_i$ iff $f(s_i) = s_i \vee g(s_i) \neq s_i$. Also by definition of feasibility,

$$\forall r \in \Delta(g(s_i),s_i) \mid s_i < \delta(s_i, r) \leq s_i \vee g(s_i) = f(s_i),\ i=0,...$$

so that $s_{i+1} \leq f(s_i)$, i=0,... Finally, since $\mathbf{LFP}_{\leq,w}(f)$ is w^+-finite, by Theorem 4 (b) there exists a finite integer $j \geq 0$ such that $s_j = \mathbf{LFP}_{\leq,w}(f)$. Hence, $f(s_j) = s_j$, which implies that $g(s_j) \leq s_j$ so that $\Delta(g(s_j), s_j) = \{\}$, since Δ is a workset function. Consequently, code (15) halts after j iterations of the **while** loop. ■

In many applications Transformation 2 is more convenient than Transformation 1, because the inflationary condition $w \leq w \vee g(w)$ holds automatically for inductive functions $f(s) = s \vee g(s)$. We need only to check the monotonicity of $s \vee g(s)$, which is guaranteed when $g(s)$ is monotone. Transformation 2 can be used to derive the reachability solution as presented earlier in Example 1.

The similarity in the programs resulting from applying Transformations 1 and 2 suggests that under some conditions, functions $f(s)$ and $s \vee f(s)$ have the same fixed point. Indeed, we can reformulate Theorem 4.1 of Cousot and Cousot [18] without their condition that T be complete:

Theorem 5: Let $(T,\leq)$ be a join semilattice. Let f: $T \to T$ be a monotone computable function. Let

$$c_1 = \textbf{LFP}(w \vee f(s),s)$$
$$c_2 = \textbf{LFP}_{\leq,w}(s \vee f(s),s)$$
$$c_3 = \textbf{LFP}_{\leq,w}(f)$$

Then

1. If c_2 is defined, then c_1 is defined and $c_1 = c_2$;
2. If c_1 is defined and T satisfies *DCC*, then c_2 is defined;
3. If $w \leq f(w)$, then $w \vee f(s)$ and f have the same set of fixed points that are greater than or equal to w, and thus $c_1 = c_3$ when they are defined.

Proof: We prove 2 only. For 1 and 3 see Cousot and Cousot [18].

We show that c_1 is a fixed point of $h(x) = x \vee f(x)$, and any other fixed point of h that is greater than or equal to w is also greater than or equal to c_1. Since $c_1 = w \vee f(c_1)$, then $c_1 \geq w$, and $c_1 = c_1 \vee f(c_1)$. Let $g(x) = w \vee f(x)$. Since f is monotone, so is g. If $s \in T$ and $s = s \vee f(s) \geq w$, then $s \geq g(s) \geq g^2(s) \geq ...$ by monotonicity of g. Since T satisfies *DCC*, then there exists an integer $k \geq 0$ such that $g^k(s) = g(g^k(s))$. Hence, $g^k(s)$ is a fixed point of g belonging to T. Since c_1 is the least fixed point of g belonging to T, then $s \geq g^k(s) \geq c_1$. Thus, c_1 is the least fixed point of h that is greater than or equal to w, and $c_2 = c_1$. ■

Example 10: We give an example where c_3 is defined but c_2 is not. Let $T = \{ 1 - 2^{-n}:\ n = 0, 1, 2,...\} \cup \{ 1 + 2^{-n}:\ n = 0, 1, 2,...\}$, let

$$f(1 - 2^{-n}) = 1 - 2^{-(n+1)}, \qquad n = 0, 1, ...$$
$$f(1 + 2^{-n}) = 1 + 2^{-(n+1)}, \qquad n = 1, 2, ...$$
$$f(2) = 2$$

Let $\leq$ be the numerical ordering of real numbers. Then $\forall\ a, b \in T$, $a \vee b = \textbf{max}(a, b)$. Let $w = 0$. Then f is monotone and $w < f(w)$. It is easy to check that $c_3 = 2$. But for any $s = 1 + 2^{-n}$, $n \geq 0$, $s = s \vee f(s)$. Thus, c_2 is not defined. This example does not contradict Theorem 5, because T does not satisfy *DCC*. ■

Theorem 5 provides an opportunity to choose between three equivalent specifications c_1, c_2, and c_3. Of these, c_1 is often the most desirable choice, because it can lead to greatly simplified preprocessing. For example, in set theoretic applications, where the empty set is **0**, application of Transformations 1 or 2 to c_1 results in the simple initializing statement p := {}. Consequently, the code introduced by finite differencing just before such an initializing statement can be as little as half the preprocessing code introduced for specifications c_2 or c_3.

Example 11: (Graph reachability continued) In Example 1 the set of vertices s that are reachable from w was given by the following specification:

$$\textbf{the}\ s:\ \ w \subseteq s \ |\ \ s = s \cup e[s] \quad \textbf{minimizing}\ s$$

which is the same as

$$(16) \quad \textbf{LFP}_{\subseteq,w}(s \cup e[s],\ s)$$

According to Theorem 5, specification (16) is equivalent to

$$(17) \quad \textbf{LFP}(w \cup e[s],\ s)$$

Making use of the same feasible functions,

$$(18) \quad \Delta(f(s),\ s) = f(s) - s$$
$$\delta(s,\ z) = s\ \textbf{with}\ z$$

as in Example 6 but relative to a more desirable function $f(s) = w \cup e[s]$ at {}, we can apply Transformation 1 to implement specification (17) with the following program:

$$
\begin{array}{ll}
(19) & p := \{\} \\
& (\textbf{while}\ \exists\ x \in ((w \cup e[p]) - p)\ ) \\
& \quad p\ \textbf{with:=}\ x \\
& \textbf{end}
\end{array}
$$

The preprocessing code that would result from applying finite differencing to program (19) is much simpler than the code that would arise from applying finite differencing to program (5). ■

The following example with nested application of fixed point transformations illustrates a more substantial benefit to preprocessing.

Example 12: (Interval partitioning) A *flow graph e* is a digraph with a unique entry node *entry* from which there are paths to every vertex in the graph. An *interval* is a smallest subgraph of *e* that contains a unique entry point *h*, called a *header* node, and also contains each node *x* in the graph whenever it contains all predecessors of *x*. Given a flow graph *e* with entry node *entry*, we want to partition its nodes into the unique set *ints* of intervals. We can specify the interval *intof*(*head*) with header node *head* as follows:

$$
\begin{array}{ll}
(20) & \textbf{the}\ int\colon \{head\} \subseteq int \mid (\forall\ x \in e[int] \mid e^{-1}\{x\} \not\subseteq int\ \textbf{or}\ x \in int) \\
& \textbf{minimizing}\ int
\end{array}
$$

which can first be transformed into,

$$\textbf{LFP}_{\subseteq, \{head\}}(int \cup \{x \in e[int] \mid e^{-1}\{x\} \subseteq int\}, int)$$

and then by Theorem 5 into,

$$\textbf{LFP}(\{head\} \cup \{ x \in e[int] \mid e^{-1}\{x\} \subseteq int\}, int)$$

Finally, using feasible functions (18) relative to function

$$f(int) = \{head\} \cup \{x \in e[int] \mid e^{-1}\{x\} \subseteq int\}$$

at {}, we can apply Transformation 1 to obtain the following **while**-loop:

$$
\begin{array}{ll}
(21) & int := \{\} \\
& (\textbf{while}\ \exists\ z \in ((\{head\} \cup \{x \in e[int] \mid e^{-1}\{x\} \subseteq int\}) - int) \\
& \quad int\ \textbf{with:=}\ z \\
& \textbf{end}
\end{array}
$$

If we define function

$$g(ints) = \{intof(entry)\} \cup intof[\cup/\{e[int] - int\colon int \in ints\}]$$

then the unique interval partition of *e* can be specified $\textbf{LFP}(g)$[9]. Once again using feasible functions (18) but relative to function *g* at {}, we can apply Transformation 1 to obtain the following implementation:

$$
\begin{array}{ll}
(22) & ints := \{\} \\
& (\textbf{while}\ \exists\ z \in ((\{intof(entry)\} \cup \\
& \quad\quad intof[\cup/\{e[int] - int\colon int \in ints\}]) - ints)) \\
& \quad ints\ \textbf{with:=}\ z \\
& \textbf{end}
\end{array}
$$

However, if we redefine the feasible functions Δ and δ differently so that

$$
\begin{aligned}
\Delta(g(ints), ints) &= intof^{-1}[g(ints)] - \cup/ints \\
&= (\{entry\} \cup e[\cup/ints]) - \cup/ints\ ^{10} \\
\delta(ints, z) &= ints\ \textbf{with}\ intof(z)
\end{aligned}
$$

then application of Transformation 1 yields,

---

[9]Since function *intof* maps a vertex into a set of vertices that forms an interval, the image of a set of vertices under *intof* is a partition of intervals.

[10]This definition is valid, because each interval has a unique header.

```
(23)    ints := {}
        (while ∃ z ∈ ({entry} ∪ e[∪/ints]) - ∪/ints))
            ints with:= intof(z)
        end
```

Finite differencing and data structure selection will transform (22) or (23) into a program that runs in time $O(\#e)$, but (23) is better in the sense that it contains only one reference to *intof* inside the **while**-loop. Because of another reference to *intof* within the **while**-loop predicate of code (22), finite differencing would introduce a whole bunch of code (which would be absent in code that results from (23)) to compute the interval with header *entry* as part of the preprocessing code before the **while**-loop. ■

An important generalization of the inductive form just discussed is

(24) $$f(x) = \vee_{i=1}^{k} g_i(x)$$

where g_i: $T \to T$, i=1,...,k.

Transformation 3:

Let $(T, \leq, \vee)$ be a join semilattice with a unique minimum element **0**. Consider functions g_i: $T \to T$, i=1,...,k, such that $f = \vee_{i=1}^{k} g_i$ is a monotone computable function. Let $w \in T$ and $w \leq f(w)$. If $\mathbf{LFP}_{\leq,w}(f)$ is w^+-finite and Δ_i and δ_i are feasible relative to g_i at every element x, $w \leq x \leq \mathbf{LFP}_{\leq,w}(f)$, i=1,...,k, then the following transformation is correct:

```
        p:= LFP≤,w(∨_{i=1}^k gi)
⇒
(25)    p := w
        (while ∃i=1,...,k ∃ z ∈ Δi(gi(p), p))
            p := δi(p,z)
        end
```

Proof: The successive values assigned to p in program (25) form a sequence s_i, i=0,..., which is strictly increasing as long as $\exists\, j = 1,...,k \mid g_j(s_i) \not\leq s_i$, which is true iff $f(s_i) = \vee_{i=1}^{k} g_i\,(s_i) \not\leq s_i$. Also by feasibility, we know that $\exists\, j = 1,...,k \mid s_{i+1} \leq g_j(s_i)$, which implies that $s_{i+1} \leq f(s_i)$ for i = 0,... Finally, since $\mathbf{LFP}_{\leq,w}(f)$ is w^+-finite, by Theorem 4 (b) there exists a finite integer $j \geq 0$ such that $s_j = \mathbf{LFP}_{\leq,w}(f)$. Hence, $f(s_j) = s_j$, which implies that $g_i(s_j) \leq s_j$ so that $\Delta_i(g_i(s_j), s_j) = \{\}$, since Δ_i is a workset function, i = 1,..,k. Consequently, code (25) halts after j iterations of the **while** loop. ■

A further refinement of Transformation 3 can be used to determine minimum solutions to systems of equations and is discussed in greater depth in Section 5. The basic idea is stated briefly as follows. Let $(T_i, \leq_i, \mathbf{0}_i)$ be a poset with minimum element $\mathbf{0}_i$, i=1,...,k. Let $T = \times_{i=1}^{k} T_i$ be the product poset with minimum element $[\mathbf{0}_1,...,\mathbf{0}_k]$. If x and y belong to T, then $x \leq y$ iff $x_i \leq_i y_i$, i=1,...,k. Let $w \in T$, and consider k monotone functions f_i: $T \to T_i$, where $f_i(w) \geq_i w(i)$, i=1,...,k. Define k corresponding monotone functions g_i: $T \to T$, i=1,...,k, as follows:

$$g_i(x)(j) = \begin{cases} f_i(x), & \text{if } j = i \\ \mathbf{0}_j, & \text{otherwise} \end{cases}$$

Transformation 3 can then be used to find the least fixed point of

(26) $$f(x) = [f_1(x),...,f_k(x)] = \vee_{i=1}^{k} g_i(x)$$

Choose $S = T$ and the following feasible functions relative to each g_i at every $x \in T$, where $x \geq w$, i=1,...,k:

```
for all p, q ∈ T where p ∨ q is defined,
    Δ(q, p)) = {} if q ≤ p, and {p ∨ q} otherwise;
    δ(p, p ∨ q) = p ∨ q,
```

Assuming that all of the posets $(T_i, \leq_i, \mathbf{0}_i)$, i = 1,...,k, are the same, we obtain the 'chaotic' iteration described by

Tenenbaum [77] and Cousot and Cousot [17] (restricted to finite iteration) to compute least fixed points of systems of equations $x_i = f_i(x_1,...,x_k)$, i=1,...,k; that is,

```
p := w
(while ∃i=1,...,k | f_i(p) > p(i) )
    p(i) := f_i(p)
end
```

More generally, if we redefine Δ and δ in the following way,

$$\Delta(q,p) = \{[i,x]: i = 1,...,k,\ p(i) < x \le (q \vee p)(i)\}$$
$$\delta(p,[i,x]) = [p(1),..,p(i\text{-}1),\ x,\ p(i\text{-}1),...,p(k)]$$

we can replace code (25) appearing in Transformation 3 by,

```
(27)  p := w
      (while ∃[i,x] ∈ Δ(f(p), p))
          p(i) := x
      end
```

We can also obtain a somewhat narrower but, perhaps, more convenient iteration than (27). If workset Δ_i and increment δ_i are feasible relative to functions g_i at all points $x \in T$, $x \ge_i w$, $i = 1,...,k$, then the following functions Δ and δ are feasible relative to f at all points $x \in T$, $x \ge w$,

(28) $\forall\ [x_1,...,x_k],\ [y_1,...,y_k] \in T,$

$$\Delta([x_1,...,x_k],\ [y_1,...,y_k]) = \{[i,t]: i = 1,...,k,\ t \in \Delta_i(x_i,y_i)\}$$

$$\delta([x_1,...,x_k],\ [i,t]) = [x_1,...,x_{i-1},\ \delta_i(x_i,t),\ x_{i+1},...,x_k]$$

Functions (28) lead to a variant of Transformation 3 in which code (25) is replaced by the code just below:

```
(29)    x := w
        (while ∃ [i, z] ∈ Δ(f(x), x))
                x_i := δ_i(x_i, z)
        end
```

Thus we have

Transformation 4: The w^+-finite solution of system (26) can be computed by (29). ■

Solving systems of equations leads to efficient solutions to two special functions. We can compute **LFP**($g(x,...,x)$) for monotone functions g, where parameter x occurs m times, by substituting m distinct identifiers x_i, i=1,...,m, for the different occurrences of x within g and finding the minimum solution to the system of m equations $x_i = g(x_1,...,x_m)$, i=1,..,m. Each variable x_i has the same solution, which is **LFP**($g(x,...,x)$). For composition of two monotone functions h and g, the solution to **LFP**($h \circ g$) is the same as the solution to variable s when we take the minimum solution to the two equations, $t = g(s)$ and $s = h(t)$.

Transformation 3 can also be used to compute the least common fixed point greater than or equal to w of a family F of functions, denoted by $\mathbf{LFP}_{\le,w}(F)$.

Theorem 6: Let $(T,\le,\vee,\mathbf{0})$ be a semilattice and $w \in T$. Let g_i: $T \to T$, i=1,...,k, be a family F of monotone, inflationary, computable functions. Let h be the composition of these k functions in any order, and let $g = \vee_{i=1}^{k} g_i$. If T has an *ACC*, then the least common fixed point $\mathbf{LFP}_{\le,w}(F)$ exists, and $\mathbf{LFP}_{\le,w}(F) = \mathbf{LFP}_{\le,w}(g) = \mathbf{LFP}_{\le,w}(h)$.

Proof: Since each function g_i, i=1,...,k, is monotone, inflationary, and computable, so is g. Then, by Theorem 3 and Corollary 2, $p = \mathbf{LFP}_{\le,w}(g)$ is defined and can be computed. Since $p = g(p) \ge g_j(p) \ge p$, j=1,...,k, then p is also a common fixed point of F. Clearly any common fixed point of F that is greater than or equal to w is also a fixed point of both g and h. Let $q \ge w$ be a fixed point of h. Without loss of generality, suppose that $h = g_k \circ g_{k-1} \circ ... \circ g_1$. Because the functions belonging to F are all inflationary, $q \le g_1(q) \le g_2(g_1(q)) \le ... \le h(q) = q$. Hence, q is a common fixed point of F. ■

Based on Theorem 6 we can show that two seemingly different classical methods of global program analysis are identical. Kildall [44] introduced a fairly general method for program analysis using iterative schema (27). Kildall's algorithm was later refined by Tenenbaum [77] and Kam and Ullman [42]. Cousot and Cousot used a strategy similar to this algorithm called 'chaotic' iteration for defining least fixed points as the limit of a sequence, and they applied it in new settings [16, 15].

Let e represent the edges of a program control flow graph with nodes labelled 1, ..., k (cf. Example 5). For each flow graph node j = 1, ..., k, let x_j be a variable storing some program fact at node j, and let g_j be a monotone flow function defined at node j. In the flow analysis frameworks of Tenenbaum and Kam and Ullman, the goal is to solve the least fixed point of the following system of equations:

(30) $x_i = \vee_{j \in e\{i\}} g_j(x_j),\ i = 1, \ldots, k$

According to Theorem 5, the least fixed point of system (30) can be rewritten equivalently as the least fixed point of

$x_i = x_i \vee (\vee_{j \in e\{i\}} g_j(x_j)),\ i = 1, \ldots, k$

which is the same as the least fixed point of

(31) $x_i = \vee_{j \in e\{i\}} (x_i \vee g_j(x_j)),\ i = 1, \ldots, k$

Finally, by Theorem 6, the least fixed point of the system of equations (31) is equivalent to the least fixed point of the following system

(32) $x_i = x_i \vee g_j(x_j),\ i = 1, \ldots, k,\ j \in e(i)$

which is precisely the system that Kildall used.

The next example illustrates a more interesting application of Theorem 6 to algorithm derivation.

Example 13: (Many function coarsest partition problem) The Many Function Coarsest Partition Problem inputs a finite set s, an initial partition P of s, and a family of total functions h_i on s, i=1,...,k. It outputs the coarsest (i.e., greatest) refinement Q of P such that $\forall\ b \in Q\ \forall\ i=1,\ldots,k\ \exists\ d \in Q \mid h_i[b] \subseteq d$. It is straightforward to reformulate this problem as computing the greatest common fixed point (that is a refinement of P) of the family of functions

$g_i(Q) = \{b \cap h_i^{-1}[q]:\ b \in Q,\ q \in Q \mid b \cap h_i^{-1}[q] \neq \{\}\}$

i=1,...,k. Since each function g_i is monotone and deflationary, the solution is to compute **GFP**($P \wedge (\wedge_{i=1}^{k} g_i)$) according to the dual forms of Theorem 6 and Theorem 5.

Let $f = P \wedge (\wedge_{i=1}^{k} g_i)$. The first step in a derivation of an efficient algorithm is to apply the dual form of Transformation 3 with the following feasible functions relative to f at P:

$\Delta(f(Q),\ Q) = Q - f(Q)$

and

$\delta(Q,q) = (Q - \{q\}) \cup \{q - b, b\}$
where $b \in \{x \in f(Q) \mid x \subset q$ **and** $\#x \leq \#q/2\}$

This leads to the algorithm just below,

```
(33)     Q := {s}
        (while ∃ q ∈ (Q - f(Q)) ∃ b ∈ f(Q) | b ⊂ q and #b ≤ #q/2)
          Q := (Q - {q}) ∪ {q - b, b}
        end
```

Finite differencing can be used to improve code (33) by preserving the invariant $P = f(Q)$ appearing in the condition of the **while**-loop. If we define for i=1,...,k,

$$split_i(P,b) = \{t \cap h_i^{-1}[b] : t \in P \mid t \cap h_i^{-1}[b] \neq \{\}\} \cup \{t - h_i^{-1}[b] : t \in P \mid t - h_i^{-1}[b] \neq \{\}\}$$

then the improved code is

```
Q := {s}
(while ∃ q ∈ (Q - P) ∃ b ∈ P | b ⊂ q and #b ≤ #q/2)
   (for i = 1,...,k)
      P := split_i(P,b)
   end
   Q := (Q - {q}) ∪ {q - b, b}
end
```

Further applications of finite differencing and appropriate data structuring will yield an algorithm that runs in $O(nk\mathbf{log}n)$ time with $O(n)$ auxiliary space. Note that Hopcroft's algorithm had a similar time bound but required $\Omega(nk)$ space in the worst case [38].

We can also consider an alternative derivation that leads to an algorithm with the same asymptotic worst case time but with fewer split operations and $O(nk)$ space. Using the method of parameterized functions, we can rewrite the arguments of function f with k distinct parameters, so that

$$f(Q_1,...,Q_k) = P \wedge (\wedge_{i=1}^{k} g_i(Q_i))$$

and solve the resulting systems of equations using feasible functions as before. The result appears just below:

```
(34)      (for i=1,...,k)
             Q_i := {s}
          end
          (while ∃ i=1,...,k ∃ q ∈ (Q_i - f(Q_1,...,Q_k))
                        ∃ b ∈ f(Q_1,...,Q_k) | b ⊂ q and #b ≤ #q/2)
             Q_i := (Q_i - {q}) ∪ {q - b, b}
          end
```

As in the previous derivation, code (34) can be improved by using the *split* functions to preserve the invariant $P = f(Q_1,...,Q_k)$; that is,

```
(35)      (for i=1,...,k)
             Q_i := {s}
          end
          (while ∃ i=1,...,k ∃ q ∈ (Q_i - P)
                        ∃ b ∈ P | b ⊂ q and #b ≤ #q/2)
             P := split_i(P,b)
             Q_i := (Q_i - {q}) ∪ {q - b, b}
          end
```

Further applications of finite differencing and data structure selection are straightforward to complete the derivation. The resulting algorithm was suggested in [58] and comes closer to Hopcroft's original algorithm [38]. Gries gave a more complete but lower level top-down (almost transformational) proof of Hopcroft's algorithm [31]. ■

Computing fixed points for functions of the form $f(w,s) = \mathbf{LFP}_{\geq,w}(g(s,t),t)$ are discussed later.

3.4. Special Data Types

It is also worthwhile to refine Transformation 1 with respect to different data types. We have already given examples of fixed point computations on set lattices, partition lattices, function posets, and so forth. In this section we consider a simple hierarchy of semilattices and corresponding methods for computing fixed points.

Let $(L, \leq, \mathbf{0})$ be a join semilattice with a unique minimum element $\mathbf{0}$. A non-$\mathbf{0}$ element $a \in L$ is called an *atom* if $\forall\, b \in L$, $b \leq a$ implies $b = \mathbf{0}$ or $b = a$. For example, in the powerset lattice over a finite set s, the atoms are the singleton sets $\{x\}$, for all x belonging to s. In the lattice of partitions over a finite set s (cf. Example 8), the atoms are the partitions

$$\{\{x\}: x \in (s-\{a, b\})\} \cup \{\{a,b\}\},$$

for all doubleton sets $\{a,b\} \subseteq s$.

For each element $w \in L$, we define the *decomposition* of w to be

$$\mathbf{dec}(w) = \{a \leq w \mid a \text{ is an atom}\}.$$

If for all elements $w \in L$,

$$w = \vee/\mathbf{dec}(w)$$ [11]

then we say that L is *decomposable*. The Shortest Path Problem in Example 7 is defined on a nondecomposable lattice.

If L is decomposable, then the following properties hold for all $a, b \in L$:

1. $a \leq b \Leftrightarrow \mathbf{dec}(a) \subseteq \mathbf{dec}(b)$
2. $a \vee b = \vee/(\mathbf{dec}(a) \cup \mathbf{dec}(b))$

If L is decomposable and $\mathbf{dec}(w)$ is finite for all $w \in L$, then we say that L is *finitely decomposable* (abbr. FD). Kildall's Constant Propagation Problem from Example 5 illustrates a decomposable lattice that is not finitely decomposable because of the maximum element $\mathbf{I}$.

If L is FD, then the following additional property holds:

3. $a \wedge b = \vee/(\mathbf{dec}(a) \cap \mathbf{dec}(b))$

and therefore L is a lattice. We also know that

4. $\mathbf{dec}(a \wedge b) = \mathbf{dec}(a) \cap \mathbf{dec}(b)$

For FD lattices we can define difference,

5. $a - b = \vee/(\mathbf{dec}(a) - \mathbf{dec}(b))$

with the property that

6. $(a \wedge b) \vee (a - b) = a$

For FD lattices the set theoretic arbitrary selection operation $\ni$ generalizes to atom selection. Of particular importance, if a is an atom and $a \notin \mathbf{dec}(T)$, then $T \vee a > T$. However, if a is an atom and $a \in \mathbf{dec}(T)$, it does not follow that $T - a < T$.

If L is FD and the dual of property 4 holds; i.e.,

7. $\mathbf{dec}(a \vee b) = \mathbf{dec}(a) \cup \mathbf{dec}(b)$,

then for any finite set A of atoms, we have

$$\begin{aligned} &\mathbf{dec}(\vee/A) \\ =\ &\cup/\{\mathbf{dec}(t): t \in A\} \\ =\ &\cup/\{\{t\}: t \in A\} \\ =\ &A \end{aligned}$$

Consequently, each element belonging to L is uniquely representable by the join of a finite set of atoms. Such a lattice is said to be *uniquely finite decomposable* (abbr. UFD). The partition lattice used in Example 8 is FD but not UFD.

For UFD lattices, difference $a - b$ defined in (5.) above is the unique minimum element d such that $(a \wedge b) \vee d = a$, and if a is an atom and $a \in \mathbf{dec}(T)$, then $T - a < T$. UFD lattices also have the following property,

[11] We define $\vee/\{\} = \mathbf{0}$.

8. $\mathbf{dec}(a - b) = \mathbf{dec}(a) - \mathbf{dec}(b)$

If L is a UFD lattice and $A = \{a \in L \mid a$ is an atom of $L\}$, it follows from the preceding discussion that **dec** is an isomorphic map from $(L, \leq, \vee, \wedge, -, \mathbf{0})$ to $(\{x \subseteq A \mid x$ is finite$\}, \subseteq, \cup, \cap, -, \{\})$; and for any set A, $(\{x \subseteq A \mid x$ is finite$\}, \subseteq, \cup, \cap)$ forms a UFD lattice. Thus, we have the following useful transformation for UFD lattices:

Transformation 5: Let f: $T \rightarrow T$ be a monotone computable function defined on UFD lattice $(T, \leq)$ with minimum element $\mathbf{0}$. Let $w \in T$ such that $w \leq f(w)$. If $\mathbf{LFP}_{\geq,w}(f)$ is w^+-finite, then the following transformation is correct:

$$
\begin{array}{ll}
 & p \ := \mathbf{LFP}_{\geq, w}(f) \\
 & \Rightarrow \\
(36) & p \ := \mathbf{0} \\
 & (\mathbf{while}\ \exists\ z \in \mathbf{dec}((w \vee f(p)) - p) \\
 & \qquad\qquad p \ \vee:= z \\
 & \mathbf{end}
\end{array}
$$

■

Powerset lattices are important examples of UFD lattices. If s is a finite set and $T = \mathbf{pow}(s)$, then functions

$$
\begin{array}{lll}
(37) & \Delta(A, B) = A - B & \text{(set difference)} \\
 & \delta(A, x) = A \ \mathbf{with}\ x & \text{(element addition)}
\end{array}
$$

are feasible at $w \in T$ relative to any monotone computable function f: $T \rightarrow T$ that is inflationary at w. An inductive form of Transformation 5 (as in Transformation 2) justifies the treatment of the reachability problem found in Example 11 and interval partitioning found in Example 12.

Transformation 5 is exceedingly useful for efficient computation of least fixed points of functions defined on UFD lattices. If we assume that $f(p)$ can be computed in polynomial time and each atom uses $O(1)$ space, then all specifications to which Transformation 5 may be applied can be implemented in a most rudimentary way with running times exponential in the size of the search space $\mathbf{dec}((\vee/\mathbf{range}\ f) \vee w)$. Transformation 5 effectively translates these exponential time specifications into procedural forms with greedy strategies and polynomial running times. Since p grows one element at a time, if the value of the expression $\mathbf{dec}((w \vee f(p)) - p)$ does not change dramatically from iteration to iteration, then it may be possible to compute the new value of $\mathbf{dec}((w \vee f(p)) - p)$ from its old value more efficiently than to compute it from scratch. Transformation 5 also provides an accurate upper bound on the iteration count for the **while**-loop in code (36). The bound is $\#\mathbf{dec}(p)$ at the final value of p. Such complexity information provided by Transformation 5 is exploited in [11] to develop a syntactic characterization of a class of set theoretic fixed point expressions that can be computed in linear time and space with respect to the input/output space.

To compute greatest fixed points in UFD lattices, we can apply the following dual form of Transformation 5:

Transformation 6: Let f: $T \rightarrow T$ be a monotone computable function defined on UFD lattice $(T, \leq)$ with maximum element $\mathbf{1}$. Let $w \in T$ such that $w \geq f(w)$. If $\mathbf{GFP}_{\leq,w}(f)$ is w^--finite, then the following transformation is correct:

$$
\begin{array}{ll}
 & p \ := \mathbf{GFP}_{\leq, w}(f) \\
 & \Rightarrow \\
(38) & p \ := \mathbf{1} \\
 & (\mathbf{while}\ \exists\ z \in \mathbf{dec}(p - (w \wedge f(p))\ ) \\
 & \qquad\qquad p \ -:= z \\
 & \mathbf{end}
\end{array}
$$

■

Transformation 6 can be used to derive an efficient cycle testing algorithm (cf. Example 3).

Now consider a general FD lattice L. In this case Transformation 6 does not apply, because the iterative atom deletion step in code (38) may make no progress. Nevertheless, we can sometimes map greatest fixed points into equivalent least fixed points that can be computed using Transformation 5.

Suppose an FD lattice L has a maximum element **1**. A non-**1** element a $\in L$ is called a *dual atom* if $\forall\, b \in L$, $b \geq a$ implies $b = \mathbf{1}$ or $b = a$. For each element $w \in L$, we define the *dual decomposition* of w to be

$$\mathbf{ddec}(w) = \{a \geq w \mid a \text{ is a dual atom}\}.$$

We say that L is *dual-decomposable*, if for all elements $w \in L$,

$$w = \wedge/\mathbf{ddec}(w).$$

If L is a lattice, then the dual lattice L' is formed from L by reversing the ordering $\leq$ (where the reverse ordering is denoted by $\leq'$), and interchanging meet and join and **1** and **0**. In reasoning about the dual lattice we would also interchange **dec** and **ddec**, and w^--finite and w^+-finite.

If L is FD and dual decomposable, then L' is FD also. Hence, we have

> **Transformation 7**: Let f: $T \rightarrow T$ be a monotone computable function defined on a dual decomposable FD lattice $(T, \leq)$ with minimum element **0** and maximum element **1**. Let $w \in T$ such that $w \geq f(w)$. If $\mathbf{GFP}_{\leq,w}(f)$ is w^--finite, then $\mathbf{GFP}_{\leq,w}(f) = \mathbf{LFP}_{\leq',w}(f)$. Hence, we can compute the greatest fixed point by applying Transformation 5 to the equivalent least fixed point in the dual lattice.

■

Although Transformation 5 still applies for FD lattices, the potentially excessive number of atoms contained in $\mathbf{dec}((w \vee f(p)) - p)$, appearing in code (36), can often make this transformation too costly. For instance, in the lattice of partitions over an n-element set s (cf. Example 8), the maximum element $\{s\}$ can be formed from the join of n-1 atoms, even though $\#\mathbf{dec}(\{s\}) = n(n\text{-}1)/2$. Fortunately, we can sometimes overcome this problem using a simple, fairly general data compression technique.

Let A be the set of atoms in an FD lattice L. For any $x \in L$ and $t \subseteq A$, we say that t is a *representation* of x if $x = \vee / t$. A function r: $L \rightarrow pow(A)$ is a *representation function* of L, if $\forall\, x \in L$, $r(x)$ is a representation of x. If r is a representation function of L, then we can obtain feasible functions

(39) $\Delta(a, b) = r(a) - \mathbf{dec}(b)$

and

(40) $\delta(b, q) = b \vee q$

relative to any monotone computable function f: $L \rightarrow L$ that is inflationary at $w \in L$. Consequently, when $\#r(x)$ is much smaller than $\#\mathbf{dec}(x)$, we can replace the set $\mathbf{dec}((w \vee f(p) - p)$ within code (36) with $r(w \vee f(p)) - \mathbf{dec}(p)$ in order to obtain better performance out of Transformations 5. Definitions (39) and (40) can also be used to improve Transformations 1, 2, 3, and 4.

To illustrate both the use of representation functions and Transformation 7, consider greatest fixed point computations on the lattice L of partitions over a finite set s (cf. Examples 8, 9, and 13). Observe that L is dual decomposable and FD, but not UFD.

If we define $datm(x) = \{x, s - x\}$ for all $x \subseteq s$, then the set of atoms in the dual lattice L' is

$$\{datm(x)\colon\ x \subseteq s \mid x \neq \{\} \ \mathbf{and}\ x \neq s\}$$

If we also define $r(X) = \{datm(x)\colon x \in X\}$ for all $X \in L$, then r is a representation function for L', since $X = \wedge/r(X)$.

In order to solve greatest fixed points in L by computing least fixed points in L', we reformulate feasible functions (39) and (40) for L' as follows,

$$r(B_1) - \mathbf{ddec}(B_2)$$
$$= \{datm(b_1): b_1 \in B_1 \mid (\exists\ b_2 \in B_2 \mid b_1 \cap b_2 \neq \{\} \textbf{ and } b_2 - b_1 \neq \{\})\}$$

which leads to the following simplified feasible functions:

$$(41) \quad \Delta(B_1, B_2) = \{b_1 \in B_1 \mid (\exists\ b_2 \in B_2 \mid b_1 \cap b_2 \neq \{\} \textbf{ and } b_2 - b_1 \neq \{\})\}$$

$$\delta(B_2, b) = B_2 \wedge datm(b) = \{x \cap b: x \in B_2 \mid x \cap b \neq \{\}\ \} \cup \{x - b: b \in B_2 \mid x - b \neq \{\}\ \}$$

relative to any monotone computable function f: $L' \rightarrow L'$ that is inflationary at $w \in L'$. If $B_2 \geq B_1$, then functions (41) further simplify to

$$(42) \quad \Delta(B_1, B_2) = \{b \in B_1 \mid b \notin B_2\}$$

$$\delta(B_2, b) = \{x \in B_2 \mid x \cap b = \{\}\ \} \cup \{b\} \cup \{x - b: x \in B_2 \mid b \subseteq x\}$$

Example 14: (Relational coarsest partition problem) Let Q be a partition of the finite set s, e be a binary relation over s, and $a \subseteq s$. Q is *stable* with respect to a if for all blocks $b \in Q$, either $b \subseteq e^{-1}[a]$ or $b \cap e^{-1}[a] = \{\}$. Q is stable if it is stable with respect to each of its blocks. Let Q_0 be an initial partition. Then the *Relational Coarsest Partition Problem* is to find the maximum stable partition $Q \leq Q_0$.

Let $\#s = n$ and $\#e = m$. Two algorithms are presented in [58]: one is a general algorithm with $O(mn)$ time complexity, and the other uses the 'smaller half' strategy to achieve a lower time complexity $O(m\mathbf{log}n)$. Both of these two algorithms can be formally derived from abstract specifications. If $f(Q)$ is the coarsest refinement of Q that is stable with respect to each block of Q, then the solution to the relational coarsest partition problem can be specified as

$$(43) \quad \mathbf{GFP}_{\leq, Q_0}(f)$$

from which the $O(mn)$ general algorithm can be derived.

Let $g(Q) = f(Q) \wedge Q_0$. We can incorporate a 'smaller half' strategy like the one used in Example 8 into the following feasible functions relative to g at Q_0 based on (42):

$$\Delta(g(Q), Q) = \{b_1 \in g(Q) \mid (\exists\ b_2 \in Q \mid b_1 \subseteq b_2 \textbf{ and } \#b_1 \leq \#b_2/2)\}$$

$$\delta(Q, b) = Q \wedge datm(b)$$

When the preceding functions are used in connection with Transformation 7, we can derive the $O(m\mathbf{log}n)$ algorithm described in [58]. ■

To compute fixed points of functions defined on lattices that are not FD, we offer no general method other than what has been suggested in the previous section on special functions or on the classical iteration (13) implied by Corollary 2. For example, fixed point computation for functions defined on the real numbers, which is certainly nondecomposable, is a whole subject outside the scope of this paper. Whenever a lattice is partly FD, in the sense that some of the lattice elements can be represented by a finite join of atoms, then we can still often employ the techniques discussed in this section. The next example illustrates this idea.

Example 15: In the constant propagation algorithm given by Reif and Lewis [61, 80], each assignment statement A is associated with either *bottom* (means undefined), *top* (means non-constant), or a real number that can result from the execution of A. Let $\mathbf{R}$ denote the set of real numbers. Under the ordering,

bottom $\leq'$ any real number $\leq'$ *top*;
real numbers are incomparable

the set $\{bottom, top\} \cup \mathbf{R}$ forms a lattice in which

$bottom \vee x = x$
$top \vee x = top$
$x \vee y = top \quad \forall\ x, y \in \mathbf{R} \mid x \neq y$

Let *val* be the set of all statement-value pairs for the given program. Let *assign* be the set of assignment statements of the given program. Then *val* is a total function from *assign* to $\{bottom, top\} \cup \mathbf{R}$. Let val_1 and val_2 be two such functions. We define $val_1 \leq val_2$ iff $val_1(s) \leq' val_2(s)\ \forall\ s \in assign$. Under this ordering, all such functions form a lattice F. The maximum function in this lattice is $val1 = \{[s,top]: s \in assign\}$ and the minimum is $val0 = \{\ [s, bottom]: s \in assign\}$. $\forall\ s \in assign$ and $v \in \mathbf{R}$, the function $\{[s,v]\} \cup \{[x, bottom]: x \in assign \mid x \neq s\}$ is an atom. Because of *top*, lattice F is decomposable but not FD. But we can still use rule (28) to obtain the following feasible functions:

$$\Delta(val_1, val_2) = \{[s, v] \in val_1 \mid val_1(s) \npreceq val_2(s)\}$$

and

$$\delta(val, [s,v]) = (val - \{[s, val(s)]\}) \cup \{[s, val(s) \vee v]\} \qquad \blacksquare$$

relative to any monotone computable function $f: F \rightarrow F$ and any element $w \in F$ where f is inflationary.

4. Fixed Point Recomputation

The preceding section showed that efficient computation of least fixed points depends, in large part, on finite differencing to avoid costly recomputation of expressions embedded within feasible functions Δ and δ. In this section we investigate application of finite differencing to fixed point expressions themselves, e.g.,

$$(44) \quad g(u,w) = \mathbf{LFP}_{\leq, w}(f(s,u), s)$$

whose input parameters u and w can be modified. We also show how to compute fixed points of functions $g(u, w)$ and, hence, nested fixed points.

In general, the need for recomputing problems arises naturally in several contexts. For example, the Cornell Synthesizer [76] is a syntactic editing system that uses an attribute grammar to implement program semantics. Whenever a program is modified using the synthesizer, the program's semantic information must be updated to reflect the editing changes. The attribute reevaluation algorithm of Reps, *et. al* [63] is an efficient incremental algorithm in the sense that it recomputes the new semantics from the old in an optimal way - performing asymptotically better than an algorithm that just recomputes the new semantics from scratch.

Incremental algorithms can also be used to solve repeated subtasks of a problem efficiently. For example, a selection sort, which repeatedly performs a linear time search for the minimum value of a set, can be turned into a faster heap sort by using a 'dynamic' heap data structure to compute the minimum set value with only a *log* factor cost each time [55].

A third context is where a problem $P(s)$ can be solved incrementally by computing P at an initial point s_0 and then at successive points $s_i = g(s_{i-1})$, $i = 1, .., n$, where g is an inexpensive incremental calculation and $s_n = s$.

Finite differencing and stream processing [28] are program transformations that provide a formal basis for studying the three kinds of recomputations just described. However, previous investigations avoided consideration of fixed point expressions. In this section we extend that earlier work in finite differencing by presenting rules for efficient recomputation of fixed point expressions (44).

4.1. Finite Differencing

In this section we give a brief introduction to *finite differencing* [57, 55], a technique that can improve the performance of programs generated by our fixed point transformations. The basic goal of this technique is to replace direct calculations of costly expressions $f(x_1,...,x_m)$ in a program region B by less expensive incremental calculations. We explain the technique by example after first presenting some definitions and convenient notational conventions.

If a variable e always stores the value of an n-variate function $f(x_1, ..., x_m)$ at a program point p, we say that equality $e = f(x_1, ..., x_m)$ is *invariant* at p. Consequently, any occurrence of expression $f(x_1, ..., x_m)$ at p is said to be *redundant* and can be replaced by variable e. Let dx_j be a modification to a variable x_j on which f depends. The *predifference* and *postdifference* of e with respect to dx_j, denoted by $\partial^- e \langle dx_j \rangle$ and $\partial^+ e \langle dx_j \rangle$ respectively, are two single-entry single-exit code blocks with the following properties.

1. If $e = f(x_1, ..., x_m)$ is invariant just before executing code

   (45) $\partial^-\ e\ \langle dx_j \rangle$
   dx_j
   $\partial^+\ e\ \langle dx_j \rangle$

   it is invariant immediately after (45) is executed.

2. The predifference and postdifference code blocks can only modify variable e and variables local to these blocks.

Consider a collection of equalities $e_j = f_j$, $j = 1,...,n$, in which each expression f_j depends only on variables $v_1,...,v_k,e_1,...,e_{j-1}$. Suppose we want to maintain and exploit all of these equalities as invariants within a single-entry program region B. The *differential* of $e_1, ..., e_n$ with respect to B, denoted by $\partial\ \{e_1, ..., e_n\}\ \langle B \rangle$, is a new code block formed from B by recursively applying the following rules.

1. Replace each modification dx occurring in B with

   (46) $\partial\ \{e_2,...,e_n\}\ \langle \partial^- e_1 \langle dx \rangle\ dx\ \partial^+ e_1 \langle dx \rangle \rangle$

   where no new occurrences of f_1 are introduced within difference code associated with invariants $e_2,...,e_n$. Also, all occurrences of f_1 appearing in $\partial^- e_1 \langle dx \rangle\ dx\ \partial^+ e_1 \langle dx \rangle$ must be redundant and are replaced by e_1. If the preceding conditions are met, we refer to e_1 as a *minimal invariant* for the differential $\partial\{e_1,...,e_n\}\langle dx \rangle$.

2. Substitute all occurrences of f_j by e_j, j=1,...,n, within the rest of B.

Based on the differential, we obtain the following general chain rules for collective predifference and postdifference code blocks:

$$\partial^-\ \{e_1, ..., e_n\}\ \langle dx_j \rangle = \partial\ \{e_2,...,e_n\}\ \langle \partial^-\ e_1\ \langle dx_j \rangle \rangle\ \ \partial^- \{e_2,...,e_n\}\ \langle dx_j \rangle$$

and

$$\partial^+\ \{e_1, ..., e_n\}\ \langle dx_j \rangle = \partial^+ \{e_2,...,e_n\}\ \langle dx_j \rangle\ \ \partial\ \{e_2,...,e_n\}\ \langle \partial^+\ e_1\ \langle dx_j \rangle \rangle$$

where e_1 is minimal.

Example 16: Two simple examples of difference code are illustrated below.

1. Let $c = 5x$. Then

   $\partial^-\ c\ \langle\ x\ {+:=}\ 1\ \rangle\ =\ c\ {+:=}\ 5$
   $\partial^+\ c\ \langle\ x\ {+:=}\ 1\ \rangle\ =\ \lambda$ [12]

[12]We use λ to denote the empty code block

Since the predifference code for c does not involve x, it could also be regarded as postdifference code; i.e.,

$$\partial^- \; c \;<\; x \;+:=\; 1 \;> \quad = \quad \lambda$$
$$\partial^+ \; c \;<\; x \;+:=\; 1 \;> \quad = \quad c \;+:=\; 5$$

2. Let $c = \{\; x \in s \mid k(x) \;\}$ where k is a computable boolean expression that does not depend on s. Then

```
∂- c < s with:= z >
=
if k(z) then
          c with:= z
end
```

and

$$\partial^+ \; c \;<\; s \;\textbf{with}{:=}\; z \;> \quad = \quad \lambda$$

For this example, also, the predifference code could just as easily be shifted to postdifference code.

We can see that, in general, difference code is not unique. ■

Difference code for products like $5x$ are included in the strength reduction transformations of most optimizing compilers and are used to replace certain products with less expensive sums [12]. Difference code for a more general class of expressions including the set former above are central to finite differencing and can be found in [22, 26, 57]. The following example adopted from [53] shows how finite differencing is used in combination with the fixed point transformations to yield efficient programs:

Example 17: (Graph reachability continued) The program derived in Example 11 contains an expensive expression $w \cup e[p] - p$ in the **while**-loop. However, its costly computation can be avoided by applying finite differencing.

Bottom up parsing decomposes this expression as follows:

$$e_1 = e[p]$$
$$e_2 = w \cup e_1$$
$$e_3 = e_2 - p$$

Next, we replace the **while**-loop in (19) by its collective differential

```
∂ {e1, e2, e3} <(while ∃ z ∈ (w ∪ e[p] - p) )
                          p with:= z
              end >
```

which is equivalent to

```
(while ∃ z ∈ e3)
      ∂ {e1, e2, e3} <p with:= z>
end
```

Among the three invariants e_1, e_2, and e_3, we know that e_1 is minimal, and within the remaining two, e_2 is minimal. Therefore we have

```
∂ {e1, e2, e3} <p with:= z>
=
∂ e3 < ∂ e2 <∂ e1 <p with:= z> > >
=

∂ e3  < ∂ e2  < (for x ∈ e{z})
                        e1 with:= x
                 end
                 p with:= z  >  >
```

```
=

∂ e3 < (for x ∈ e{z})
            e2 with:= x
            e1 with:= x
        end
        p with:= z   >
=

(for x ∈ e{z})
        if x ∉ p then
                e3 with:= x
        end
        e2 with:= x
        e1 with:= x
end
e3 less:= z
p with:= z
```

The initialization code is simple:

```
e1 := {}
e2 := w
e3 := w
```

Consolidating all of the preceding code, we obtain the following procedure for computing graph reachability:

```
e1 = {}
e2 = w
e3 = w
p := {}
(while ∃ z ∈ e3)
    (for x ∈ e{z})
        if x ∉ p then
              e3 with:= x
        end
        e2 with:= x
        e1 with:= x
    end
    e3 less:= z
    p with:= z
end
```

Analysis for useless code determines that variables e_1 and e_2 are never used. After all assignments to e_1 and e_2 are eliminated, we obtain the following much improved code:

```
(47) e3 = w
     p := {}
     (while ∃ z ∈ e3)
        (for x ∈ e{z})
          if x ∉ p then
              if x ∉ e3 then
                  e3 with:= x
              end
          end
        end
        e3 less:= z
        p with:= z
     end
```

Note that in the final phase, we guarded the operation e_3 **with**:= x with the condition $x \notin e_3$ so that x is always added to set e_3. This program can be implemented with suitable data structures to have a time and space complexity linear to the size of the input graph. ■

4.2. Incremental Recomputation of Fixed Points

We now derive difference code for the fixed point expression

$g(u, w) = \mathbf{LFP}_{\leq, w}(f(s,u), s)$

with respect to modifications to u and w. In Section 3 we showed different ways of computing g according to properties of f. In the next lemma we also show how properties of g, which are useful in deriving difference code for g, depend on properties of f.

Lemma 7: Let f: $S \times T \to S$ be a computable function monotone in each of its arguments, where (S, $\leq$) and (T, $\leq$) are posets, and $w \in S$. Consider function g: $T \times S \to S$ with the rule $g(u, w) = \mathbf{LFP}_{\leq,w}(f(s,u),s)$. Then

1. $g(u, w)$ is monotone and inflationary in w.
2. If $g(u, w)$ is defined for $[u, w] \in T \times S$, then for any $w' \in S$ such that $w \leq w' \leq g(u, w)$, $g(u, w')$ is also defined, and $g(u, w') = g(u, w)$.
3. $g(u, w)$ is monotone in u over the subset of T for which $w \leq f(w, u)$ and $g(u, w)$ is w^+-finite.
4. If $(S, \leq) = (T, \leq)$, and $f(s,u)$ is inflationary in u for all $s \in S$, then $g(u, w)$ is inflationary in u.

Proof:

1. Suppose that $w, w' \in S$, $w' \geq w$, and $g(u, w)$, $g(u, w')$ are defined. By definition, $g(u, w)$ is the smallest solution to $f(s, u) = s$ greater than or equal to w. Since $g(u, w') \geq w' \geq w$ and since $g(u, w')$ is another solution to $f(s, u) = s$ greater than or equal to w, then $g(u, w') \geq g(u, w)$.
2. $g(u, w)$ is the smallest $x \in S$ satisfying the conditions $w' \leq x$ and $x = f(x, u)$.
3. Let $f_u(s) = f(s, u)$. Let $u, u' \in \{x \in T \mid w \leq f_x(w)$ **and** $g(x,w)$ is w^+-finite$\}$ and $u \leq u'$. Then $f_u(x) \leq f_{u'}(x)\ \forall\ x \in S$. Since $f_u(s)$ and $f_{u'}(s)$ are monotone in s, then $f_u^k(x) \leq f_{u'}^k(x)\ \forall\ x \in S$, $\forall\ k = 0, 1, \ldots$. Hence, for some integer $k \geq 1$, $g(u, w) = f_u^k(w) \leq f_{u'}^k(w) = g(u', w)$ by Corollary 2 and Theorem 3(3).
4. If $g(u,w) = s'$, then $s' = f(s',u) \geq u$. ■

Based on Lemma 7, we can modify the least fixed point $g(u, w)$ incrementally with respect to modifications in u and w. In the following discussion, we use x_{old} and x_{new} to represent the value of the variable x before and after modifications respectively, and use $f_y(x)$ to represent the function of x defined by $f(x, y)$.

Theorem 8: Let $p = g(u, w)$ be defined as in Lemma 7. Assume that $\forall\ [x, y] \in T \times S$, $g(x, y)$ is y^+-finite. Let δ_1 and δ_2 be increment functions.

1. Let Δ and δ be functions feasible relative to $f_{u_{new}}$ at p_{old}. If $w \leq f(w, u_{old})$, then

$\partial^+\ p$ <u := $\delta_1(u,z)$>
=

(48) (**while** $\exists\ x \in \Delta(f(p,u), p)$)
    $p := \delta(p,x)$
  end

The predifference code is empty. (Note that postdifference and not predifference code is used, because (48) references the new values of u.)

2. If $(S, \leq, \vee)$ is a join semilattice, $p_{old} \vee w_{new} \leq f(p_{old} \vee w_{new}, u)$, Δ' and δ' are functions feasible relative to w_{new} at p_{old}, and Δ and δ are functions feasible relative to f_u at $p_{old} \vee w_{new}$, then

$\partial^+ \; p \; \langle w \; := \; \delta_2(w, z) \rangle$
$=$

(49) $(\textbf{while} \; \exists \; y \in \Delta'(w, p))$
$\quad p \; := \; \delta'(p, y)$
end
$(\textbf{while} \; \exists \; x \in \Delta(f(p,u), \; p))$
$\quad p \; := \; \delta(p, x)$
end

The predifference code is empty.

Proof:

1. Because of the feasibility of Δ and δ, and the y^+-finiteness of $g(x, y)$ for all $[x, y] \in T \times S$, the successive values assigned to p in the **while**-loop forms a sequence $s_0, s_1, ..., s_k$ with $s_0 = p_{old} = g(u_{old}, w)$ and $s_k = g(u_{new}, p_{old})$. We need to show that $g(u_{new}, p_{old}) = g(u_{new}, w)$. Since f is monotone in u, and $u_{new} = \delta_2(u_{old}, z) \geq u_{old}$, then $f(w, u_{new}) \geq f(w, u_{old}) \geq w$. Since for all $x \in T$, $g(x, w)$ is w^+-finite, then by Lemma 7(3) $w \leq p_{old} \leq g(u_{new}, w)$. Therefore $g(u_{new},w) = g(u_{new},p_{old})$ by Lemma 7(2).
2. The proof is similar to 1. $\forall \; [x, y] \in T \times S$, the condition that $g(x, y)$ is y^+-finite implies that $g(x, y)$ is defined, and thus $y \leq g(x, y)$. By Lemma 7(1), $g(u, w_{old}) \leq g(u, w_{new})$. Thus $w_{new} \leq p_{old} \vee w_{new} \leq g(u, w_{new})$, where $p_{old} = g(u, w_{old})$. Then by Lemma 7(2), $g(u, w_{new}) = g(u, p_{old} \vee w_{new})$, which is computed by (49). Note that at the end of the first **while**-loop of (49), $p = p_{old} \vee w_{new}$, and the condition $p \leq f(p, u)$ is satisfied. ■

It is interesting to consider various ways in which code (48) and (49) can be improved. One obvious approach to speed up this code is to avoid computing $\Delta(f(p, u), p)$ by maintaining equality $e = \Delta(f(p, u), p)$ as an invariant together with p. In the case of (48) the chain rule applied to both p and e leads to the following difference code blocks:

$$\partial^- \; \{e, p\} \; \langle u \; := \; \delta_1(u,z)\rangle \; = \; \partial^- \; e \; \langle u \; := \; \delta_1(u,z)\rangle$$

and

$$\partial^+ \; \{e, p\} \; \langle u \; := \; \delta_1(u,z)\rangle \; = \; \partial^+ \; e \; \langle u \; := \; \delta_1(u,z)\rangle \quad \partial \; e \; \langle \partial^+ \; p \; \langle u \; := \; \delta_1(u,z)\rangle\rangle$$

Under the same conditions, a similar improvement is possible for (49).

One drawback with code (48) is that it can only be used as postdifference code (if we are to avoid copying parameter u), since it references the new value of u. Such inflexibility is undesirable, because it can preclude opportunities for further optimization. However, we can overcome this problem whenever the collective postdifference code $\partial^+ \; \{e, p\} \; \langle u := \delta_1(u,z)\rangle$ involves no occurrences of u. This fortunate situation arises when $\partial^+ \; e \; \langle u := \delta_1(u,z)\rangle$ is empty and $\partial \; e \; \langle \partial^+ \; p \; \langle u := \delta_1(u,z)\rangle\rangle$ involves no free occurrences of u. In this case we can use an empty postdifference block $\partial^+ \; \{e, p\} \; \langle u := \delta_1(u,z)\rangle$ and the following predifference code:

$$\partial^- \; \{e, p\} \; \langle u \; := \; \delta_1(u,z)\rangle \; = \; \partial^- \; e \; \langle u \; := \; \delta_1(u,z)\rangle \quad \partial \; e \; \langle \partial^+ \; p \; \langle u \; := \; \delta_1(u,z)\rangle\rangle$$

(50) $= \; \partial^- \; e \; \langle u \; := \; \delta_1(u,z)\rangle$
$(\textbf{while} \; \exists \; x \in e)$
$\quad \partial \; e \; \langle p \; := \; \delta(p,x)\rangle$
end

Since code (50) involves no occurrences of u, it can be used as either pre- or postdifference code.

We can exploit contexts where predifference code (50) is correct in two ways:

1. by uncovering alternative ways of computing (44) and

2. by space optimizations.

If δ: $T \times X \rightarrow T$ is an increment function, $u_0 \subseteq X$ is a set such that u can be computed by executing

```
u := 0
(for z ∈ u0)
      u := δ(u, z)
end
```

then one alternative way to establish $p = g(u, w)$ is by the following incremental calculation involving a search through u

```
∂- {e, p} <u := 0>
(for z ∈ u0)
        ∂- {e, p} <u := δ(u, z)>
end
```

This implementation may be useful when a search through u is unavoidable because of global computational requirements. Note that a similar incremental approach to compute $g(u, w)$ based on a search through w and using predifference code (49) is also possible.

In computing nested expressions such as $g(h(x), w)$ or in maintaining the value of such expressions across modifications to x, we can often avoid evaluation or maintenance of subexpression $h(x)$. To see how this is achieved, consider the three equalities $u = h(x)$, $p = g(u, w)$, and $e = f(p, u) - p$. If we can calculate $u = h(x)$ by executing

```
∂- u <x := 0>
(for z ∈ x0)
        ∂- u <x := δ(x, z)>
end
```

then we can also calculate $p = g(h(x), w)$ by executing the following block:

```
∂- {e, p, u} <x := 0>
(for z ∈ x0)
        ∂- {e, p, u} <x := δ(x, z)>
end
```

Whenever the difference code blocks ∂^- $\{e, p, u\}$ $<x := \mathbf{0}>$ and ∂^- $\{e, p, u\}$ $<x := \delta(x, z)>$ contain no occurrences of u except for modifications to u, then maintenance of the equality $u = h(x)$ is unnecessary; all modifications to u can be eliminated. Goldberg and Paige called this technique *vertical loop fusion* [28].

The following theorem and corollary can sometimes be used to introduce opportunities for vertical fusion.

Theorem 9: Let $(L, \leq, \mathbf{0})$ be a meet semilattice with a unique minimum element $\mathbf{0}$; let f: $L \rightarrow L$ be a computable monotone function; let $c \in L$ be a constant such that $f(s \wedge c) \wedge c = f(s) \wedge c$. If $\mathbf{LFP}(f)$ is $\mathbf{0}$-finite, then $\mathbf{LFP}(f) \wedge c = \mathbf{LFP}(f(s) \wedge c, s)$.

Proof: Let $h(s) = f(s) \wedge c$. We prove by induction that for all integers $i \geq 1$, $h^i(\mathbf{0}) = f^i(\mathbf{0}) \wedge c$. For $i = 1$, it is trivial. Assume this holds for $i < k$, where $k > 1$. Then

$$
\begin{aligned}
& h^k(\mathbf{0}) \\
= \; & h(h^{k-1}(\mathbf{0})) \\
= \; & f(f^{k-1}(\mathbf{0}) \wedge c) \wedge c \\
= \; & f^k(\mathbf{0}) \wedge c
\end{aligned}
$$

■

Corollary 10: Let $(L, \leq, \mathbf{0})$ be a meet semilattice with a unique minimum element $\mathbf{0}$; let f, g: $L \rightarrow L$ be two computable monotone functions. If $\mathbf{LFP}(f)$ and $\mathbf{LFP}(g)$ are $\mathbf{0}$-finite, and $f(s \wedge \mathbf{LFP}(g)) \wedge \mathbf{LFP}(g) = f(s) \wedge \mathbf{LFP}(g)$, then $\mathbf{LFP}(f) \wedge \mathbf{LFP}(g) = \mathbf{LFP}(f \wedge \mathbf{LFP}(g))$. ■

Example 18: (Sink-source problem) Consider a directed graph represented as a binary edge relation e. Let *sources* and *sinks* be two sets of nodes. We want to find out all the nodes in *e* occurring within any path from *sources* to *sinks*. Let

$$f1(s) = sources \cup e[s]$$
$$f2(s) = sinks \cup e^{-1}[s]$$

Then we can solve this problem by first computing the set

$$reach = \mathbf{LFP}(f1)$$

of all nodes reachable from *sources* and the set

$$access = \mathbf{LFP}(f2)$$

of all nodes that can reach *sinks*. Then the required solution is simply

$$output = reach \cap access$$

It is easy to verify

$$f1(s \cap access) \cap access = f1(s) \cap access$$

and

$$f2(s \cap reach) \cap reach = f2(s) \cap reach$$

which leads to the new specifications

$$(51) \quad output = \mathbf{LFP}(f1(s) \cap access) = \mathbf{LFP}(f2(s) \cap reach)$$

by corollary 10. Either of these specifications is desirable, since it can be computed in one pass through *e*.

■

Example 19: (Elimination of dead code and unreachable code) A program statement is called *dead* if it makes no contribution to the output of the program; a statement is called *unreachable* if it cannot be reached along any path in the flow graph of the program beginning from the entry statement. Based on Theorem 9 we can derive code that eliminates both dead code and unreachable code in one pass.

Consider a program representation in which variable *prints* is the set of print statements, variable *iuses* maps each program statement to the variable uses it contains, *usetodef* is a binary relation that maps each variable use to the variable definitions that can reach it, *instof* is a function mapping each variable occurrence to the statement immediately enclosing it, and *compound* is a function that maps each statement to the compound statement immediately enclosing it (i.e., **if** or **while** statement). In [52] an efficient dead code elimination procedure was derived from the following specification:

$$(52) \quad live = \mathbf{LFP}(f_3)$$

where $f_3(s) = prints \cup instof[usetodef[iuses[s]]] \cup compound[s]$, which determines live statements from data and control flow considerations.

Unreachable statements can be eliminated by applying the graph reachability algorithm of Example 1 to the program flow graph:

$$(53) \quad reachable = \mathbf{LFP}(f_4)$$

where $f_4(s) = \{entry\} \cup succ[s]$, *entry* is the only entry statement of the program, and $succ\{x\}$ is the set of all the possible successors of statement *x*. The set of statements that are both *live* and reachable is

$$useful = live \cap reachable$$

The reader can verify that

$$f_3(s \cap reachable) \cap reachable = f_3(s) \cap reachable$$

Hence, by Theorem 9, we have

$$useful = \mathbf{LFP}(f_3(s) \cap reachable,\ s)$$

from which a one pass program can be derived. Note that we do not have

$$useful = \mathbf{LFP}(f_4(s) \cap live,\ s)$$

because in general

$$f_4(s \cap live) \cap live \neq f_4(s) \cap live$$ ■

4.3. Recomputation of Fixed Points of Distributive Functions

Although we can recompute least fixed points incrementally under some conditions, the decremental recomputation of least fixed points is much more difficult. In [43], Kaplan and Ullman specify their solution to a general weak type analysis problem as $\mathbf{GFP}(\Psi\circ\Phi(s))$, where $\Psi(s) = \mathbf{LFP}(s \wedge B(x), x)$, $\Phi(s) = \mathbf{LFP}(s \wedge F(x), x)$, and B and F are two monotone functions representing backward and forward type analysis respectively. They compute this greatest fixed point as the limit of the decreasing sequence $\mathbf{1} = s_0, s_1, ...,$ where $s_{i+1} = \Psi\circ\Phi(s_i)$ is recomputed from scratch for each s_i in this sequence. We have also failed to find a general efficient decremental method for computing least fixed points.

However, least fixed points can still be recomputed efficiently under some restricted conditions. One special case observed in [46] is when a function has a unique fixed point. In this case, the fixed point can be treated as both a least and a greatest fixed point, and thus can be recomputed incrementally and decrementally.

In this section we show that if f is *distributive*, i.e., $f(x \vee y) = f(x) \vee f(y)$, then the following expression

$$(54) \quad g(w) = \mathbf{LFP}(w \vee f(x),\ x)$$

can be modified efficiently when w is incremented or decremented. We assume that f is a monotone computable function on a finite *UFD* lattice $(L, \leq, \vee, \mathbf{0})$ (cf. Section 3.4). We show that this problem is reducible to the problem of graph reachability.

Let $r = r(w, e)$ be the set of vertices s reachable from a source set w along paths in a directed graph e. We have shown that r is the least fixed point of the function $f_w(x) = w \cup e[x]$.

Lemma 11: If the directed graph e is acyclic, then the function $f_w(x)$ has a unique fixed point.

Proof: Let $x_0 = \mathbf{LFP}(f_w)$, and x_1 be any fixed point of f_w. Then $x_0 \subseteq x_1$. To show also that $x_1 \subseteq x_0$, let $y = x_1 - e[x_1]$. Since $x_1 = w \cup e[x_1]$, then $y \subseteq w$. Since e is acyclic, then x_1 is the set of vertices reachable from y along paths in e. Therefore $x_1 = \mathbf{LFP}(f_y)$. Since by Lemma 7(3) $\mathbf{LFP}(f_t)$ is monotone in t, then $x_1 \subseteq x_0$. ■

Therefore, by Theorem 8(1) and its dual, if e is acyclic, we have

$$\partial^+ \quad r \ \langle w \ \mathbf{with} := z\rangle$$
$$=$$
$$(55) \quad (\mathbf{while}\ \exists\ t \in (w \cup e[r] - r))$$
$$\qquad r\ \mathbf{with} := t$$
$$\mathbf{end}$$

and

$$\partial^+ \ r \ \langle w \ \mathbf{less} := z\rangle$$
$$=$$
$$(56) \quad (\mathbf{while}\ \exists\ t \in (r - (w \cup e[r])))$$
$$\qquad r\ \mathbf{less} := t$$
$$\mathbf{end}$$

Note that the inflationary condition of Theorem 8(1) and the deflationary condition of its dual are satisfied, since all functions are inflationary at $\mathbf{0}$ and deflationary at $\mathbf{1}$.

The next theorem shows how the efficiency of (55) and (56) can be further improved by finite differencing.

Lemma 12: If $r^+ = r_{new} - r_{old}$, $r^- = r_{old} - r_{new}$, $e^+ = \{[x, y] \in e \mid x \in r^+\}$, and $e^- = \{[x, y] \in e \mid x \in r^-\}$, then we can compute code (55) in $O(1 + \#e^+)$ steps and the code (56) in $O(1 + \#e^-)$ steps.

Proof: Let $new1 = w \cup e[r] - r$, $new2 = r - w \cup e[r]$, $numpred = \{[x, \#(e^{-1}\{x\} \cap r)]: x \in$ **domain** $e \cup$ **range** $e\}$. Analysis of the following two difference code blocks yields the result.

```
∂+ {r, new1, numpred} <w with:= z>
=
if z ∉ r then
     new1 with:= z
end
(while ∃ t ∈ new1)
   (for x ∈ e{t})
     numpred(x) +:= 1
     if x ∉ r then
        new1 with:= x
     end
   end
   new1 less:= t
   r with:= t
end
```

and

```
∂+ {r, new2, numpred} <w less:= z>
=
if numpred(z) = 0 then
    new2 with:= z
end
(while ∃ t ∈ new2)
   (for x ∈ e{t})
     numpred(x) -:= 1
     if numpred(x) = 0 and x ∉ w then
        new2 with:= x
     end
   end
   new2 less:= t
   r less:= t
end
```

■

Let C be the set of strongly-connected components for a directed graph e. It is well known [73] that the directed graph $G = \{[u, v] \in C \times C \mid (u \neq v)$ **and** $(\exists [m, n] \in e \mid m \in u$ **and** $n \in v)\}$ is acyclic. For all vertices $v \in$ **domain** $e \cup$ **range** e, let $component(v)$ be the strongly-connected component that contains v. Then the set C and map *component* can be computed in $O(\#e)$ time [73] (see also [3]).

Hence, we can conclude,

Lemma 13: For directed graphs in general, r can be maintained in $O(1 + \#e^+)$ steps when w changes by an element addition and in $O(1 + \#e^-)$ steps when w changes by an element deletion (c.f. Lemma 12).

Proof: If $X = component[w]$, and if Y is the set of strongly-connected components reachable from X in G, then

$$r = \cup / Y$$

X can be modified in unit time when w changes by deletion or addition. To see this, let $refcount(x) = \#(x \cap w)$ for each component $x \in C$. Then

```
∂- {X, refcount} <w with:= z>
=
if refcount(component(z)) = 0 then
            X with:= component(z)
end
refcount(component(z)) +:= 1
```

$$
\begin{aligned}
&\partial^- \ \{X,\ refcount\}\ \langle w\ \textbf{less}{:=}\ z\rangle\\
&=\\
&\textbf{if}\ \ refcount(component(z)) = 1 \quad \textbf{then}\\
&\qquad\qquad X\ \ \textbf{less}{:=}\ component(z)\\
&\textbf{end}\\
&refcount(component(z))\ \text{-}{:=}\ 1
\end{aligned}
$$

Whenever X is modified in the preceding difference code, we can modify Y by the difference code given in the proof of Lemma 12. ■

Finally, consider the expression

$$g(w) = \textbf{LFP}(w \vee f(x),\ x)$$

where f is a monotone, distributive, computable function on a finite *UFD* lattice (L, $\le$, $\vee$, $\mathbf{0}$). Since L is *UFD*, we can represent w and $g(w)$ by their decompositions. Let dw = **dec**(w) and dg = **dec**($g(\vee/dw)$). We discuss how to maintain the invariant dg with respect to the modifications dw **with**:= z and dw **less**:= z, where z is an atom.

Let $W = dw$ **with 0**. Then

$$
\begin{aligned}
&\partial^-\ \ W\ \ \langle dw\ \ \textbf{with}{:=}\ z\rangle\\
&=\\
&W\ \ \textbf{with}{:=}\ z
\end{aligned}
$$

and

$$
\begin{aligned}
&\partial^-\ \ W\ \ \langle dw\ \ \textbf{less}{:=}\ z\rangle\\
&=\\
&W\ \ \textbf{less}{:=}\ z
\end{aligned}
$$

Let A be the set of all the atoms of L, and $V = A \cup \{\mathbf{0}\}$. Let $e = \{[v, u]: v \in V, u \in \textbf{dec}(f(v))\}$. Let $F(x) = w \vee f(x)$ and $F'(x) = W \cup e[x]$. We have

Lemma 14: For all $s \subseteq V$ such that $s \ne \{\}$, $F(\vee/s) = \vee/F'(s)$.

Proof:

$$
\begin{aligned}
&\ \ F(\vee/s)\\
&= w \vee f(\vee/s)\\
&= w \vee \vee/\{f(x):\ x \in s\} && \text{since } f \text{ is distributive}\\
&= w \vee \vee/\{\vee/\textbf{dec}(f(x)):\ x \in s\} && \text{since } L \text{ is } UFD\\
&= w \vee \vee/\{\vee/e\{x\}:\ x \in s\} && \text{by the definition of } e\\
&= w \vee \vee/e[s] && \text{where } e[s] = \cup/\{e\{x\}: x \in s\}\\
&= \vee/(W \cup e[s])\\
&= \vee/F'(s)
\end{aligned}
$$

■

By induction we can prove further that

Corollary 15: $F^k(w) = \vee/F'^k(W)$ for all k = 0, 1, 2, ... ■

If r = **LFP**($e[s]\cup W$, s), then by Corollary 15 **LFP**($w\vee f(x)$, x) = $\vee/r$, which implies that $dg = r$. (Remember that dg = **dec**($g(w)$) = **dec**(**LFP**($w\vee f(x)$, x)).) Since we know how to modify r with respect to the modifications W **with**:= z and W **less**:= z, then we know how to modify dg respect to the modifications dw **with**:= z and dw **less**:= z. Thus we have

Theorem 16: The invariant dg can be maintained in $O(1 + \#e^+)$ steps when dw changes by an element addition and in $O(1 + \#e^-)$ steps when dw changes by an element deletion (c.f. Lemma 12).

4.4. Nested Least Fixed Points

Efficient ways to recompute least fixed points are especially useful in contexts where least fixed points are composed with other functions. However, in the special case where the least fixed point operator is composed with itself, it is sometimes best to avoid an incremental least fixed point calculation in favor of a more direct approach described as follows.

Consider again the function

$$g(u, w) = \mathbf{LFP}_{\leq, w}(f(s,u), s)$$

where f: $S \times S \to S$ is a computable function monotone and inflationary in each of its arguments, $(S, \leq)$ is a poset, and $w \in S$. Then $g(u,w)$ is a function on S monotone in u and w and we can further consider the least fixed point of $g(u,w)$ with respect to either u or w. The following theorem tells when these fixed points are defined and how they can be computed.

Theorem 17: Let $f_1(x) = f(x,x)$. For any function $z(x, y)$, let $z_y(x) = z(x,y)$. If f: $S \times S \to S$ is a computable function monotone and inflationary in each of its arguments, $w, q \in S$, and $w \vee q$ is defined, then

1. $\mathbf{LFP}_{\leq,q}(g(u,w), w) = g(u, q)$
2. Let $a = \mathbf{LFP}_{\leq,q}(g(u,w), u)$ and $b = \mathbf{LFP}_{\leq,q\vee w}(f_1(u), u)$. Let $h_1 = \{f_1^i(w\vee q)$: $i = 0, 1, ...\}$, and $h_2 = \{g_w^i(q)$: $i = 0, 1, ...\}$.
   Then both a and b are defined and $a = b$ if either of the following conditions are satisfied:
   i. h_1 is finite;
   ii. h_2 is finite, and $\{f_x^i(w)$: $i = 0, 1, ...\}$ is finite for all $x \in h_2$.

Proof:

1. Trivial.
2. Suppose h_1 is finite. By Corollary 2, b is defined and there exists an integer $t \geq 0$ such that

$$(57)\quad \forall\ k = t,\ t+1,\ ...,\ b = f_1^k(w \vee q) = f_1^k(b)$$

Since f is monotone and inflationary in each of its arguments, we can prove by induction that

$$(58)\quad \forall\ i = 0, 1, ...,\ f_1^i(w \vee q) \leq f_b^{i+1}(w) \leq f_1^{i+1}(b)$$

From (57) and (58), we have

$$\forall\ k = t+1,\ t+2,\ ...,\ b = f_b^k(w) = \mathbf{LFP}_{\leq, w}(f(s,b),\ s) = g(b,w)$$

This means that b is a fixed point of $g_w(x)$ greater than or equal to q. On the other hand, if c is any fixed point of $g_w(x)$ greater than or equal to q, then

$$\begin{aligned}(59)\quad c &= g_w(c)\\ &= \mathbf{LFP}_{\leq, w}(f(s,c),\ s)\\ &= f(c,c)\\ &\geq w\end{aligned}$$

Thus c is also a fixed point of f_1 greater than or equal to $w \vee q$. Since b is the least fixed point of f_1 greater than or equal to $w \vee q$, we have $c \geq b$. Thus $b = \mathbf{LFP}_{\leq,q}(g_w(x)) = a$.

Suppose h_2 is finite and $\{f_x^i(w)$: $i = 0, 1, ...\}$ is finite for all $x \in h_2$. By Corollary 2, a is defined. As shown in (59), a is a also a fixed point of f_1 greater than or equal to $w \vee q$. Let $c \geq w \vee q$ be any fixed point of $f_1(x)$. We prove by induction that $c \geq g_w^t(q)$ for any $t = 0, 1, ...$. For $t = 0$, the assertion $c \geq q$ is trivially true. Assume that $c \geq g_w^i(q)$ for some integer $i \geq 0$. Then for some integer $k \geq 0$,

$$
\begin{aligned}
c &= f_1^k(c) \\
&\geq f_1^k(w \vee g_w^i(q)) && \text{by induction hypothesis} \\
&\geq f_{w \vee g_w^i(q)}^k(w \vee g_w^i(q)) && \text{since } f \text{ is monotone and inflationary} \\
& && \text{in each of its arguments} \\
&\geq f_{g_w^i(q)}^k(w) \\
&= g_w^{i+1}(q)
\end{aligned}
$$

Since $a = g_w^t(q)$ for some integer $t \geq 0$, then $c \geq a$. Hence a is the least fixed point of $f_1(x)$ greater than or equal to $w \vee q$, *i.e.*, $a = b$. ■

5. Systems of Equations

In this section we return to the topic of systems of equations introduced in Section 3.3. Let $(T_i, \leq_i, \mathbf{0}_i)$ be a poset with minimum element $\mathbf{0}_i$, i=1, ..., n. Let $T = \times_{i=1}^n T_i$ be the product poset with minimum element $\mathbf{0} = [\mathbf{0}_1, ..., \mathbf{0}_n]$. If x and y belong to T, then $x \leq y$ iff $x_i \leq_i y_i$, i=1, ..., n.

Consider the following system of equations:

$$
(60) \qquad \begin{aligned}
&\textbf{the } x_1, ..., x_n: \\
&\quad x_1 = f_1(x_1, ..., x_n), \\
&\quad ..., \\
&\quad x_n = f_n(x_1, ..., x_n) \\
&\textbf{minimizing } x_1, ..., x_n
\end{aligned}
$$

where functions f_i: $T \to T_i$, i=1, ..., n, are computable and monotone in each of their parameters. System (60) can be solved by such classical methods as Jacobi iteration, Gauss-Seidel iteration, or Gaussian elimination. But in many nonnumerical application, these approaches are not efficient. In Section 3.3 several efficient iterative methods were discussed (cf. Transformation 4).

In this section we derive potentially efficient elimination and hybrid methods. But before doing this, it is interesting to note that the iterative methods of Section 3.3 for solving systems of equations can be obtained by straightforward refinements of Transformation 1. To see this, let $X = [x_1, ..., x_n]$ and $F(X) = [f_i(x_1, ..., x_n), ..., f_n(x_1, ..., x_n)]$. Then specification (60) is equivalent to **LFP**(F). If **LFP**(F) has a **0**-finite solution, and Δ and δ are feasible relative to F at **0**, then it can be solved as follows by Transformation 1:

```
(61) X := 0
     (while ∃ Z ∈ Δ(F(X), X))
         X := δ(X, Z)
     end
```

5.1. Elimination Method

Now we derive an elimination method for solving system (60). Let $g_n(x_1, ..., x_{n-1}) = \mathbf{LFP}(f_n(x_1, ..., x_n), x_n)$. Replacing x_n in the first n-1 equations by $g_n(x_1, ..., x_{n-1})$, we get the following new system with n-1 equations and variables:

$$
(62) \qquad \begin{aligned}
&\textbf{the } x_1, ..., x_{n-1}: \\
&\quad x_1 = f_1^1(x_1, ..., x_{n-1}), \\
&\quad ..., \\
&\quad x_{n-1} = f_{n-1}^1(x_1, ..., x_{n-1}), \\
&\textbf{minimizing } x_1, ..., x_{n-1}
\end{aligned}
$$

where

$$f_i^1(x_1, ..., x_{n-1}) = f_i(x_1, ..., x_{n-1}, g_n(x_1, ..., x_{n-1}))$$

It is not difficult to show that if n-tuple $[x_1, ..., x_n]$ is the value of expression (60), then (n-1)-tuple $[x_1, ..., x_{n-1}]$ is the value of (62). Moreover, if $[x_1, ..., x_{n-1}]$ is the value of expression (62), then $[x_1, ..., x_{n-1}, g_n(x_1, ..., x_{n-1})]$ is the value of (60).

Applying this elimination step repeatedly, we will eventually get a one-equation system

the x_1: $x_1 = f_1^{n-1}(x_1)$ **minimizing** x_1

which can be solved by Transformation 1:

```
(63)    x1 := 01
        (while ∃ z ∈ Δ1(f1^(n-1)(x1), x1))
                x1 := δ1(x1, z)
        end
```

If B represents code block (63), then we can maintain the invariants

$x_2 = \mathbf{LFP}(f_2^{n-2}(x_1, x_2'), x_2'),$
$...,$
$x_n = \mathbf{LFP}(f_n(x_1, ..., x_{n-1}, x_n'), x_n'),$

within B according to the following differential code:

(64) $\partial\ x_n <\partial\ x_{n-1}\ ...\ <\partial\ x_2 < B > >...>$

Transformation 8: The **0**-finite solution of the system (60) can be computed by (64). ■

The nested differential (64) can be expanded using Transformation 1 and Theorem 8. For example, when n = 2, (64) can be expanded into

```
(65)    x1 := 01
        x2 := 02
        (while ∃ z2 ∈ Δ2(f2, x2))
            x2 := δ2(x2, z2)
        end
        (while ∃ z1 ∈ Δ1(f1, x1))
            x1 := δ1(x1, z1)
            (while ∃ z2 ∈ Δ2(f2, x2))
                x2 := δ2(x2, z2)
            end
        end
```

The length of the expanded code from Transformation 8 grows quickly with n. This fact limits the use of Transformation 8 as the main transformation scheme for system (60). However, a hybrid transformation, discussed in the next section, overcomes this problem by combining both the iterative method of transformation 4 and the elimination method of transformation 8.

5.2. Minimizing the Number of Worksets

Let us partition the n variables x_i, $i = 1,..,n$, appearing in the system of equations (60) into two sets

$e_1 = \{x_i: i = 1,..,p\}$

and

$e_2 = \{x_i: i = p+1,..,n\}$

By eliminating the variables in e_2 from system (60), we get a new system

```
(66)    the x1, ..., xp:
            x1 = f1'(x1, ..., xp),
            ...,
            xp = fp'(x1, ..., xp),
        minimizing x1, ..., xp
```

Applying Transformation 4 to this new system, we get the following program

(67) x_i := $\mathbf{0}_i$, $i = 1,..,p$
(**while** $\exists$ i = 1, ..., p, $\exists$ $z \in \Delta_i(f_i', x_i)$)
$x_i = \delta_i(x_i, z)$
end

Let $f_n'(x_1, ..., x_{n-1})$ = **LFP**($f_n(x_1, ..., x_n)$, x_n) and $f_i'(x_1, ..., x_{i-1})$ = **LFP**($f_{i+1}'(x_1, ..., x_i)$, x_i), $i = n$-1, ..., p+1. By preserving invariants

(68) $x_i = f_i'(x_1, ..., x_{i-1})$, $i = p+1, ..., n$

within code (67), we have the following hybrid solution for system (60):

(69) ∂ $\{x_{p+1}, ..., x_n\}$
<x_i := $\mathbf{0}_i$, $i = 1, ..., p$
(**while** $\exists$ i = 1, ..., p, $\exists$ $z \in \Delta_i(f_i, x_i)$)
$x_i := \delta_i(x_i, z)$
end>

Transformation 9: The **0**-finite solution of the system (60) can also be solved by (69). ■

If the variables in e_2 are chosen in such a way that f_i does not depend on x_j for $i = p$+1, ..., n and $j = i$, i+1, ..., n, then the invariants in (68) can be maintained without resorting to fixed point iteration. In this case, we save the time and space of maintaining the worksets Δ_i for $i = p+1, ..., n$. Furthermore, if (69) contains no occurrences of x_i for some i=p+1, ..., n except for the modifications to x_i, then the variable x_i needs not be maintained at all (cf. *vertical loop fusion*, section 4.2.)

The set e_2 with the above property can be found using Algorithm 5 below. It first creates the *dependency graph* of (60)

(70) e = {[i,j]: $i,j \in$ {1, ..., n} | f_i directly depends on x_j}

and then repeatedly chooses a node from e with no self-loop and merges it into each of its predecessors until the graph is stable. The nodes eliminated from the graph correspond to e_2.

Algorithm 5:

```
e1 := {1,..,n}
e2 := {}
e := {[i,j]: i,j ∈ {1,..,n} | fi directly depends on xj}
(while ∃ i ∈ e1 | i ∉ e{i})
        e1 less:= i
        e2 with:= i
        (for k ∈ e^-1(i))
            e less:= [k,i]
            (for t ∈ e(i))
                  e with:=  [k,t]
            end
        end
        (for t ∈ e(i))
                  e less:=  [i,t]
        end
end
```
■

Lemma 18: In Algorithm 5, let e_0 be the initial value of e, e_1' be the final value of e_1 and e_2' be the final value of e_2. Then the subgraph $e' = \{[a, b] \in e_0 \mid a, b \in e_2'\}$ is acyclic.

Proof: Let $[a_1, ..., a_k]$ be a (nonempty) cycle in e_0. Since Algorithm 5 does not delete self-loops and does not break cycles, then $\{a_1, ..., a_k\} \cap e_1' \neq \{\}$. Thus $\{a_1, ..., a_k\} \cap e_2' \neq \{a_1, ..., a_k\}$. ■

Since no worksets are needed for the variables in e_2, we want the size of e_1 to be as small as possible. But according to [27], e_1 is a *feedback set*[13] of e, and the problem of finding the minimum feedback set is *NP*-complete. Thus,

Theorem 19: The problem of minimizing the number of equations in system (60) by substitution is *NP*-complete. ■

Example 20: (Grammar transformation.) Let G be a context free grammar with grammar symbols V, terminals T, and start symbol a. For each production $x \in G$, let $lhs(x)$ be the nonterminal symbol in the left hand side of x, and let $rhs\{x\}$ be the set of grammar symbols appearing in the right hand side of x. We can apply two transformations on G:

$$trans1(G) = \mathbf{LFP}(\{x \in G \mid rhs\{x\} \subseteq T \cup lhs[s]\}, s)$$

restricts G to all productions that derive strings of terminals and

$$trans2(G) = \mathbf{LFP}(\{x \in G \mid lhs(x) \in (\{a\} \cup rhs[s])\}, s)$$

restricts G to all productions that are derivable from the start symbol a.

We can find a grammar equivalent to G, but restricted so that each production is derivable from the start symbol a and each nonterminal derives strings of terminals by first applying *trans1* and then *trans2* to G. (But not the other way around!) Thus the required grammar is simply *trans2*(*trans1*(G)). Let G_1 = *trans1*(G) and G_2 = *trans2*(G_1), then Transformation 8 gives the following program:

G_1 := {}
G_2 := {}
(**while** $\exists\ z \in (\{x \in G \mid rhs\{x\} \subseteq (T \cup lhs[G_1])\} - G_1))$
$\qquad \partial\ G_2$ <G_1 **with** := z>
end ■

More complicated grammar transformations such as those used to turn a positive context free grammar into Greibach normal form [19] can also be expressed as fixed point specifications and significant loop fusion of the same kind has been observed.

Example 21: Consider the following interprocedural analysis problem. Suppose we are given a set of procedure names *procs* of some input program P. For each procedure f, let *params*(f) be the tuple of formal parameters of f, and let *calls*{f} be the set of **call**-instructions occurring within f. For each **call**-instruction $i \in calls\{f\}$, let *called*(i), which can be either a procedure name or a parameter of f, be the procedure called by i, and let *args*(i) be the tuple of arguments passed in procedure call i. For example, suppose P contains the following procedure declaration:

proc $f(x_1, ..., x_n)$
.
.
i: **call** $g(y_1, ..., y_m)$
.
.
end

Then, $f \in procs$, $params(f) = [x_1, ..., x_n]$, $i \in calls\{f\}$, $called(i) = g$, and $args(i) = [y_1, ..., y_m]$. Note that an argument in a procedure call can be a procedure name or a procedure parameter.

From the above described information, we want to find

[13] Given a graph $G(V, A)$, a subset $V' \subseteq V$ is called a *feedback set* of G if every cycle in G includes at least one vertex form V'

1. the set *pparams*{*f*} of procedure parameters for each procedure *f*;
2. the set *assoc*{*x*, *f*} of procedures that at runtime might be associated with formal parameter *x* of procedure *f*;
3. the set *maycall*{*f*} of procedures that might call *f* at runtime;
4. the set *callsto*{*f*} of call instructions that might be made to procedure *f* at runtime.

An SQ^+ specification of the these four sets is:

$$
\begin{array}{l}
\textbf{the}\ \ maycall,\ pparams,\ callsto,\ assoc:\\
\quad maycall\ = \{[f, calls^{-1}(i)]:\ [f,i] \in callsto\},\\
\quad pparams\ = \{[f,x]:\ [x,f] \in \textbf{domain}\ assoc\},\\
\quad callsto\ = \{[called(i),i]:\ [f,i] \in calls\ |\ called(i) \in procs\} \cup\\
\qquad\qquad \{[g,i]:\ [f,i] \in calls,\ g \in assoc\{called(i),f\}\},\\
\quad assoc\ = \{[[params(g)(j),g],args(i)(j)]:\ [g,i] \in callsto,\\
\qquad\qquad j = 1,..,\#args(i)\ |\ args(i)(j) \in procs\} \cup\\
\qquad\qquad \{[[params(g)(j),g],f]:\ [g,i] \in callsto,\\
\qquad\qquad j = 1,..,\#args(i),\ f \in assoc\{args(i)(j),calls^{-1}(i)\}\},\\
\textbf{minimizing}\ \ maycall,\ pparams,\ callsto,\ assoc
\end{array}
$$

Algorithm 5 will compute e_1 = {*maycall*, *pparams*, *callsto*} and e_2 = {*assoc*}, and application of Transformation 9 will produce the following one-pass program:

$$
\begin{array}{l}
\partial\ \{maycall,\ pparams,\ callsto\}\\
\qquad <\ \ assoc := \{\}\\
\qquad\quad (\textbf{while}\ \exists\ z \in (f - assoc))\\
\qquad\qquad assoc\ \textbf{with}{:=}\ z\\
\qquad\quad \textbf{end}\ \ >
\end{array}
$$

where

$$
\begin{array}{l}
f = \{[[params(g)(j),g],args(i)(j)]:\ [g,i] \in callsto,\\
\qquad j = 1,..,\#args(i)\ |\ args(i)(j) \in procs\} \cup\\
\quad \{[[params(g)(j),g],f]:\ [g,i] \in callsto, j = 1,..,\#args(i),\\
\qquad f \in assoc\{args(i)(j),calls^{-1}(i)\}\}
\end{array}
$$

■

6. Implementation Issues

We have presented general theorems and transformations for computing and recomputing fixed points. These transformations are defined semantically in terms of program properties that are usually undecidable. In order to incorporate these transformations as part of an effective mechanical program development system, we need to define stronger syntactically defined decidable properties that imply these undecidable semantic properties. Our approach, which is similar to Sintzoff's method of valuations [71], is to specify properties using a formal system of pattern directed inductive definitions. Definitions for the following properties are essential and have been implemented within RAPTS: computable, well-typed, monotone, inflationary, and finite.

In a user-friendly implementation we would also want to use convenient specifications outside of SQ^+ (cf (6) or (7)) but transformable into SQ^+. Some of these problems will be addressed in this section.

6.1. Inflationary and Deflationary

Inflationary and deflationary conditions must be checked when we use Corollary 2 and Theorem 8. Although these properties are undecidable, a recursive class of inflationary and deflationary SQ^+expressions can be generated by composition from a predefined collection of basic inflationary and deflationary functions (see Table 6-1) by the following rules:

f	g	$f \circ g$
inflationary	**inflationary**	**inflationary**
deflationary	**deflationary**	**deflationary**

and these rules can also be implemented using an S-attributed grammar. Note that the function $f(s) = s \vee g(s)$ is always inflationary, and the function $f(s) = s \wedge g(s)$ is always deflationary.

6.2. Finiteness and Finite Chain Conditions

Our transformations and theorems also frequently require conditions such as *ACC*, *DCC*, existence of **1**, or existence of **0**. These properties can be provided in or deduced from data type declarations. For example, *ACC* and *DCC* are implied by a finite poset; the existence of **0** is implied by a *UFD* lattice.

Transformation 2 may require the condition $\#\mathbf{range}(g) < \infty$. This condition holds if g is any subset of an set theoretical expression involving only **0**, **1**, and input parameters. Compile time analysis of inclusion and membership relations have been studied before by Schwartz [68, 67].

6.3. Monotone and Antimonotone

Although monotonicity is undecidable [32], in practice we can recognize a large subclass of computable monotone and antimonotone functions as described below:

1. Define basic computable monotone and antimonotone functions as are shown in table 6-1;
2. If we know whether two functions f and g are monotone or antimonotone, then we can determine whether their composition $f \circ g$ is monotone or antimonotone by the following table:

Monotonicity of the composition function $f \circ g$

f	g	$f \circ g$
monotone	**monotone**	**monotone**
antimonotone	**monotone**	**antimonotone**
monotone	**antimonotone**	**antimonotone**
antimonotone	**antimonotone**	**monotone**

3. If $f(x, y)$ is monotone in each of its parameters x and y, then the function $g(x) = f(x, x)$ is monotone in x; if $f(x, y)$ is antimonotone in each parameter x and y, then the function $h(x) = f(x, x)$ is antimonotone in x;
4. If $f(x, y)$ is a function with a finite range, is monotone in each parameter x and y, and is inflationary in x, then the functions $g(w, y) = \mathbf{LFP}_w(f(x, y), x)$ and $h(w, y) = \mathbf{GFP}_w(f(x, y), x)$ are monotone in w and y and have finite ranges.

The preceding rules define a decidable sublanguage of computable monotone (respectively antimonotone) SQ^+ functions, which we call *positive* (respectively *negative*) functions that can be recognized, say, by an S-attributed attribute grammar (see [4]). For example, the SQ^+ function

$$f(x) = s - (t - x)$$

is recognized as positive, because it is the composition of two negative basic functions $f1(e) = s - e$ and $f2(x) = t - x$. But the monotone function

$$f(x) = x - (x \cap s)$$

is not, because we have no rules for the difference of two positive functions.

We are currently developing a transformational programming system in which these and other properties defined by inductive definitions are implemented. Our results will be reported in a subsequent paper.

Let s**,** t **be sets, and let** f **be a map (set of pairs), and let** k, p **be predicates**

Expression	Parameters that satisfying conditions			
	Monotone	**Anti-monotone**	**infla-tionary**	**deflat-ionary**
$x \in s$	s			
$y \in f\{x\}$	f			
$f\{x\}$	f			
$f[s]$	s **and** f			
f^{-1}	f			
domain f	f			
range f	f			
$\#s$	s			
$\{x \in s \mid k(x)\}$	s **and** k			s
$s \cap t$	s **and** t			s **and** t
$s - t$	s	t		s
$s \cup t$	s **and** t		s **and** t	
$s \times t$	s **and** t			
$\exists x \in s \mid k(x)$	s **and** k			
$\forall x \in s \mid k(x)$	k	s		
$\neg k$		k		
k **and** p	k **and** p			k **and** p
k **or** p	k **and** p		k **and** p	

Table 6-1: Basic monotone, antimonotone, inflationary, and deflationary functions

6.4. Equational Form

We have mentioned problem specifications that can sometimes be expressed more conveniently outside of SQ^+ in either of the two forms:

(71) **the** s: $\mathbf{0} \leq s \mid k(s)$ **minimizing** s

or

(72) **the** s: $\mathbf{1} \geq s \mid k(s)$ **maximizing** s

Before any of our fixed point transformations can be applied, we must first transform $k(s)$ into an equivalent equational form

$s = f(s)$

in which f is monotone. We have no general method to do this, but the following transformations have proved to be very useful:

1. **the** s: $\forall x \in s \mid k(x)$ **maximizing** $s \Rightarrow$ **GFP**(s - $\{x \in s \mid$ **not** $k(x)\}$, s)
2. **the** s: $\{x \in s \mid k(x)\} = \{\}$ **maximizing** $s \Rightarrow$ **GFP**(s - $\{x \in s \mid k(x)\}$, s)
3. $f(s) \leq s \Rightarrow s = s \vee f(s)$
4. $s \leq f(s) \Rightarrow s = s \wedge f(s)$
5. negation elimination and De Morgan's rules

Other useful rules for deriving normal form specifications for (71) and (72) include:

1. **the** s: $s \geq \mathbf{0}$ | $s = f(s)$ **and** $s = g(s)$ **minimizing** s
   $\Leftrightarrow$
   $\mathbf{LFP}(h_1(s) \vee h_2(s),\ s)$
   $\Leftrightarrow$
   $\mathbf{LFP}(h_2 \circ h_1)$
   $\Leftrightarrow$
   $\mathbf{LFP}(h_1 \circ h_2)$

   where $h_1(x) = \mathbf{LFP}_{\leq,\, x}(f)$ and $h_2(x) = \mathbf{LFP}_{\leq,\, x}(g)$.
2. **the** s: $s \geq \mathbf{0}$ | $s = f(s)$ **or** $s = g(s)$ **minimizing** s
   $\Leftrightarrow$
   the s: $s = \mathbf{min}(\mathbf{LFP}(f),\ \mathbf{LFP}(g))$

   where we assume that both $\mathbf{LFP}(f)$ and $\mathbf{LFP}(g)$ are defined.
3. **the** s, t: $s \geq v,\ t \geq \mathbf{0}$ | $s = f(t)$ and $t = g(s)$ **minimizing** $s,\ t$
   $\Leftrightarrow$
   $\mathbf{LFP}_{\leq,\, v}(f(g(s)),\ s)$
4. **the** s: $s \geq v_1$ and $s \geq v_2$ | $s = f(s)$ **minimizing** s
   $\Leftrightarrow$
   $\mathbf{LFP}_{\leq,\ v_1 \vee v_2}(f)$

The above rules are merely a preliminary *ad hoc* collection.

7. Conclusion

7.1. A Survey of Related Work

Earlier in the paper the important connection between fixed point computation and global program optimization was mentioned. More recently, fixed point computation has also had a strong impact on relational databases. In 1979 Aho and Ullman [2] noted that Codd’s Relational Calculus is unable to express transitive closure, and they suggested extending Relational Calculus with fixed point operators. Their paper triggered an extensive study into the expressive power of languages with fixed point constructs.

In the theoretical direction logicians have made rapid progress. In 1982 Immerman [39] and Vardi [79] proved that, in the presence of a linear order (≤), every relational query computable by a Turing Machine in polynomial time with respect to the size of its input is expressible in first order logic (*FO*) extended with a least fixed point operator (**LFP**). Conversely, every *FO*(≤) + **LFP** query is computable by a Turing Machine in polynomial time with respect to the cardinality of the given structure. Immerman also showed that any query expressible with nested fixed points can be expressed with a single least fixed point application.

One problem is that fixed point operations can be computed most easily for monotone formula, but monotonicity is undecidable [32]. Also, not every formula monotone in its parameter P is equivalent to a formula positive in P [5]. Since positivity is decidable, it is fortunate that Gurevich and Shelah [34] proved that, for every monotone formula ψ, there is a positive formula ψ' such that ψ and ψ' have the same least fixed point.

In the pragmatic direction, database researchers have been looking for efficient ways of computing the least fixed point defined by a set of recursive logical queries. They all noticed that the repeated computation of expensive expressions in the fixed point iteration is one of the main sources of inefficiency. Numerous strategies have been proposed to solve this problem.

In 1981 McKay and Shapiro [47] presented a rule-based inference system that can handle recursive rules. In their system, queries are implemented as *processes* representing nodes in a graph with edges reflecting *producer-consumer* relationships between processes. Each process consumes information from its child processes and produces information to its parent processes until the whole system reaches a fixed point. No data is transmitted more than once along the same path. While their method is AI oriented, their one-element-at-a-time strategy is quite close to the principle of our finite differencing.

In 1984, Henschen and Naqvi [37] suggested that a recursive query could be expanded into a set of nonrecursive ones that could be evaluated iteratively. The expansion terminates when no more solutions are found. Their method works well when the query dependency graph forms a single cycle, but in more general situations the control structures of the generated programs would be very complicated. In 1986, Naughton [50] showed that under some conditions, this expansion can be done at compile time.

In 1985, Ullman [78] presented a more general approach. Instead of finding a single solution, he suggested the use of different strategies (capture rules) for different queries. This idea is fully developed in the design of NAIL! [48]. One interesting strategy used in NAIL! is to divide the whole system of queries into subsystems, with each subsystem corresponding to a strongly connected component in the dependency graph. These subsystems are solved one by one in a topologic order. This strategy further reduces the number of worksets.

In 1987, Kyu-Young Whang [81] introduced a rudimentary form of fixed point iteration together with finite differencing to evaluate recursive logic queries efficiently. He noticed that when this technique is applied to linear recursive queries, a one pass algorithm can be obtained. In the same article, he also raised the question of minimizing the number of worksets.

7.2. Future Work and Conclusions

Although much work remains to be done before we can say how useful our fixed point transformations are in practice, we have presented numerous examples where they could be applied effectively to nontrivial problems, specified conveniently in SQ^+. Besides the examples presented here, we have also shown that a significant fragment of an optimizing compiler [11] and even the problem of planarity testing [10] are amenable to our transformational methodology.

Several open problems and further research arise from this work. We mention some of these briefly below:

1. Are there more general conditions under which the least and greatest fixed points can be recomputed efficiently?
2. Can our previous work characterizing a class of linear time set theoretic fixed point expressions [11] be generalized to fixed points over more general lattices and semilattices?
3. Can our transformations be used to implement Prolog queries efficiently?
4. Can they be applied to implement circular attribute grammars efficiently along the lines of Farrow [24]?

Conventional software development methodology can be derived from the following informal schema:

```
(converge)
        problem formulation;
        algorithm design;
        programming;
        debugging;
end
```

Problem formulation and algorithm design involve the most conceptual work, and programming and debugging are the most tedious. We have shown in this paper rudimentary theoretical underpinnings for a computer assisted program development system in which low level debugging is unnecessary, programming is highly simplified, algorithm design is facilitated by a small collection of powerful transformations, and the major effort is in problem specification (where errors can still be introduced).

Backus [6] has criticized conventional programming languages for *'their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs'*. We hope that our approach can overcome some of these weaknesses.

References

1. Aczel, P. An Introduction to Inductive Definitions. In *Handbook of Mathematical Logic*, Barwise, J., Ed., North-Holland, 1977, pp. 739-782.

2. Aho, A.V. and Ullman J.D. Universality of Data Retrieval Language. Proc. 6th ACM Symp. on Principles of Programming Languages, 1979, pp. 110-117.

3. Aho, A., Hopcroft, J., and Ullman, J. *Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

4. Aho, A., Sethi, R., and Ullman, J. . *Compilers.* Addison-Wesley , 1986.

5. Ajtai, M and Gurevich, Yuri. "Monotone versus Positive". *JACM 34*, 4 (Oct. 1987), 1004-1015.

6. Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". *CACM 21*, 8 (Aug 1978), 613-641.

7. Bauer, F. L. and Wossner, H. *Algorithmic Language and Program Development.* Springer-Verlag, Berlin, 1982.

8. Birkhoff, G. *Lattice Theory.* American Mathematical Society, Providence, 1966.

9. Cai, J. *Fixed Point Computation and Transformational Programming*. Ph.D. Th., Rutgers University, May 1987. appears in Rutgers Technical Report DSC-TR-217.

10. Cai, J. An iterative Version of Hopcroft and Tarjan's Planarity Testing Algorithm. Tech. Rept. 324, NYU/Courant Insitute, Oct., 1987.

11. Cai, J. and Paige, R. Binding Performance at Language Design Time. Proc. 14th ACM Symp. on Principles of Programming Languages, Munich, West Germany, January, 1987, pp. 85-97.

12. Cocke, J. and Kennedy, K. "An Algorithm for Reduction of Operator Strength". *CACM 20*, 11 (Nov 1977), 850-856.

13. Cocke, J. and Schwartz, J. T. *Programming Languages and Their Compilers.* CIMS, New York University, 1969. Lecture Notes.

14. Codd, E. "A Relational Model of Data for Large Shared Data Banks". *CACM 13*, 6 (June 1970), 377-387.

15. Cousot., P. Semantic Foundations of Program Analysis. In *Program Flow Analysis*, Muchnick, S., Jones, N., Eds., Prentice Hall, 1981, pp. 303-342.

16. Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixed points. Proc. 4th ACM Symp. on Principles of Programming Languages, Los Angeles, CA, Jan, 1977, pp. 238-252.

17. Cousot, P. and Cousot, R. "Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations". *SIGPLAN Notices 12*, 8 (Aug 1977), 1-12.

18. Cousot, P. and Cousot, R. "Constructive Versions of Tarski's Fixed Point Theorems". *Pacific J. Math 82*, 1 (May 1979), 43-57.

19. Davis, M. and Weyuker, E. *Computability, Complexity, and Languages.* Academic Press, 1983.

20. de Bakker, J. *Mathematical Theory of Program Correctness.* Prentice-Hall, 1980.

21. Dewar, R., Grand, A., Liu S. C., Schwartz, J. T., and Schonberg, E. "Program by Refinement, as Exemplified by the SETL Representation Sublanguage". *TOPLAS 1*, 1 (July 1979), 27-49.

22. Earley, J. "High Level Iterators and a Method for Automatically Designing Data Structure Representation". *Journal of Computer Languages 1*, 4 (1976), 321-342.

23. Emerson, E.A. and Lei, C. L. Model Checking in the Propositional Mu-Calculus. Tech. Rept. 86-06, University of Texas at Austin, Department of Computer Sciences, 1986.

24. Farrow R. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. Proc. ACM Symp. on Compiler Construction, 1986, pp. 85-99.

25. Feather, Martin S. A survey and classification of Some Program Transformation Approaches and Techniques. In *IFIP TC 2 WG2.1 Working Conference on Program Specification and Transformation, Bad Tolz, FRG, 1986*, L. G. L. T. Meertens, Ed., North-Holland, 1987, pp. 165-196.

26. Fong, A. and Ullman, J. Induction Variables in Very High Level Languages. Proc. 3rd ACM Symp. on Principles of Programming Languages, Jan, 1976, pp. 104-112.

27. Garey, M. and Johnson, D. *Computers and Intractability.* Freeman and Johnson, 1979.

28. Goldberg, A. and Paige, R. Stream Processing. Proc. ACM Symp. on LISP and Functional Programming, Aug, 1984, pp. 53-62.

29. Gordon, M., Milner, A., and Wadsworth, C. *Edinburgh LCF.* Springer-Verlag, 1979.

30. Gratzer, G. *General Lattice Theory.* Birkhauser, 1978.

31. Gries, D. "Describing an Algorithm by Hopcroft". *Acta Informatica 2* (1973), 97-109.

32. Gurevich, Y. Toward Logic Tailored for Computational Complexity. Lecture Notes in Math: Computation and Proof Theory, 1984, pp. 175-216.

33. Gurevich, Y. Logic and the Challenge of Computer Science. In *Current Trends in Theoretical Computer Science*, Egon Boerger, Ed., Computer Science Press, 1987.

34. Gurevich, Y. and Shelah, S. Fixed-point Extensions of First-order Logic. 26th Annual Symposium on Foundation of computer Science, 1985, pp. 346-353.

35. Harel, D. and Kozen, D. "A Programming Language for the Inductive Sets, and Applications". *Information and Control 63*, 2 (1985), 1-27.

36. Hecht, M. and Ullman, J. "A Simple Algorithm for Global Data Flow Analysis Problems". *SIAM J. Comput. 4*, 4 (December 1975), 519-532.

37. Henschen,L.J. and Naqvi,S.A. "On Compiling Queries in Recursive First-order Database". *CACM 31* (1984), 47-85.

38. Hopcroft, J. An n log n Algorithm for Minimizing States in a Finite Automaton. In *Theory of Machines and Computations*, Kohavi and Paz, Ed., Academic Press, New York, 1971, pp. 189-196. Proc. Intl. Symp. on Theory of Machines and Computation.

39. Immerman, N. Relational queries computable in polynomial time. Proc. 14th ACM Symp. on Theory of Computing, May, 1982, pp. 147-152.

40. Immerman, N. Languages Which Capture Complexity Classes. Proc. 15th ACM Symp. on Theory of Computing, April, 1983, pp. 347-354.

41. Kam, J. and Ullman, J. "Global Data Flow Analysis and Iterative Algorithms". *JACM 23*, 1 (1976), 158-171.

42. Kam, J. and Ullman, J. "Monotone Data Flow Analysis Frameworks". *Acta Informatica 7* (1977), 305-317.

43. Kaplan, M. and Ullman, J. "A Scheme for the Automatic Inference of Variable Types". *JACM 27*, 1 (1980), 128-145.

44. Kildall, G. A Unified Approach to Global Program Optimization. Proc. 1st ACM Symp. on Principles of Programming Languages, Oct, 1973, pp. 194-206.

45. Kleene. *Introduction to Meta-Mathematics.* Van Nostrand, Princeton, NJ, 1952.

46. Marlowe, T., Paull, M.C. and Ryder, B. Applicability Of Incremental Iterative Algorithms. Tech. Rept. DCS-TR-159, Rutgers University, Aug, 1985.

47. McKay, D. and Shapiro, S. Using Active Connection Graphs for reasoning with resursive rules. Proceedings 7th IJCAI, 1981, pp. 368-374.

48. Morris, K., Ullman, J. D., and Gelder, A. V. Design Overview of the NAIL! System. Tech. Rept. STAN-CS-86-1108, Stanford University, May, 1986.

49. Naqvi, S. and Henschen, L. Synthesizing Least Fixed Point Queries into Non-recursive Iterative Programs. IJCAI 83, Aug, 1983, pp. 25-28.

50. Naughton, J. F. Optimizing Function-Free Recursive Inference Rules. Tech. Rept. STAN-CS-86-1114, Stanford University, 1986.

51. O'Keefe, R.. Finite Fixed-Point Problems. Logic Programming, 1987, pp. 729-743. Proc. 4th Intl. Conf. on Logic Prog..

52. Paige, R. Transformational Programming -- Applications to Algorithms and Systems. Proc. 10th ACM Symp. on Principles of Programming Languages, Jan, 1983, pp. 73-87.

53. Paige, R. Supercompilers - Extended Abstract. In *Program Transformation and Programming Environments,* P. Pepper, Ed., Springer Verlag, 1984, pp. 331-340.

54. Paige, R., Tarjan, R., and Bonic, R. "A Linear Time Solution to the Single Function Coarsest Partition Problem". *TCS 40,* 1 (Sep 1985), 67-84.

55. Paige, R. "Programming With Invariants". *IEEE Software 3,* 1 (Jan 1986), 56-69.

56. Paige, R. and Henglein, F. "Mechanical Translation of Set Theoretic Problem Specifications Into Efficient RAM Code - A Case Study". *Journal of Symbolic Computation 4*, 2 (Aug. 1987), 207-232.

57. Paige, R. and Koenig, S. "Finite Differencing of Computable Expressions". *ACM TOPLAS 4*, 3 (July 1982), 402-454.

58. Paige, R. and Tarjan, R. "Three Partition Refinement Algorithms". *SIAM J. Comput. 16*, 6 (Dec 1987), 973-989.

59. Papadimitriou, C. "A Note On The Expressive Power of PROLOG". *Bullitin of EATCS 26* (1985), 21-23.

60. Paull, M. *Algorithm Design - A recursion Transformation Framework.* Wiley, 1988.

61. Reif, J. H. and Lewis, H. R. Symbolic evaluation and the global value graph. Proc. 4th ACM Symp. on Principle of Programming Languages. , Jan, 1977, pp. 104-118.

62. Reif, J. and Scherlis, W. Deriving Efficient Graph Algorithms. Carnegie-Mellon U., 1982. Technical Report.

63. Reps, T., Teitelbaum, T., and Demers, A. "Incremental Context-Dependent Analysis for Language-Based Editors". *ACM TOPLAS 5*, 3 (July 1983), 449-477.

64. Rogers, H., Jr. *Theory of Recursive Functions and Effective Computability.* McGraw Hill, 1967.

65. Schonberg, E., Schwartz, J. T., and Sharir, M. "An Automatic Technique for Selection of Data Representations in in SETL Programs". *ACM TOPLAS 3*, 2 (Apr 1981), 126-143.

66. Schwartz, J. T. *On Programming: An Interim Report on the SETL Project, Installments I and II.* CIMS, New York Univ., New York, 1974.

67. Schwartz, J. T. "Optimization of Very High Level Languages, Parts I, II". *J. of Computer Languages 1*, 2,3 (1975), 161-218.

68. Schwartz, J.T. "Automatic Data Structure Choice in a Language of Very High Level". *CACM 18*, 12 (Dec 1975), 722-728.

69. Scott, D. S. "Data Types as Lattices". *SIAM J. Comptng 5* (1976), 522-587.

70. Sharir, M. "Some Observations on Formal Differentiation". *ACM TOPLAS 4*, 2 (Apr. 1982), 196-225.

71. Sintzoff, M. "Calculating Properties of Programs by Valuations on Specific Models". *ACM SIGPLAN Notices 7,* 1 (1972), 203-207.

72. Suppes, P. *Axiomatic Set Theory.* Dover, 1972.

73. Tarjan, R. E. "Depth first search and linear graph algorithms". *SIAM J. Computing 1*, 2 (1972), 146-160.

74. Tarjan, R. Iterative Algorithms for Global Flow Analysis. In *Algorithms and Complexity - New Directions and Recent Results*, J. Traub, Ed., Academic Press, 1976, pp. 71-101.

75. Tarski, A. "A Lattice-Theoretical Fixpoint Theorem and its Application". *Pacific Journal of Mathematics 5* (1955), 285-309.

76. Teitelbaum, T. and Reps, T. "The Cornell Program Synthesizer: a syntax-directed programming environment". *CACM 24*, 9 (Sep 1981), pp. 563-573.

77. Tenenbaum, A. *Type Determination for Very High Level Languages*. Ph.D. Th., New York University, Dept. of Computer Science, Oct 1974. appears in Courant Computer Science Report 3.

78. Ullman, J.D. "Implementation of Logical Query Languages for Databases". *ACM TODS 10*, 3 (September 1985), 289-321.

79. Vardi, M. Complexity of Relational Query Languages. Proc. 14th ACM Symp. on Theory of Computing, May, 1982, pp. 137-146.

80. Wegman, M. N. and Zadeck, F. K. Constant Propagation with Conditional Branches. Proc. 12th ACM Symp. on Principles of Programming Languages, Jan, 1985, pp. 291-299.

81. Whang, K., and Navathe, S. B. An Extended Disjunctive Normal Form Approach for Processing Recursive Logic Queries in Loosely Coupled Environments. Proc. 13th Intl. Conf. on Very Large Data Bases, Sept., 1987, pp. 275-287.

Appendix I. Turing Machine Simulation

In 1982, Immerman [39] and Vardi [79] proved that every relational query over a finite linearly ordered set D and computable by a Turing Machine in polynomial time with respect to the cardinality of D is expressible in first order logic extended with a least fixed point operator. Papadimitriou [59] proved similar results for PROLOG. The following theorem indicates the expressive power of SQ^+.

Theorem 20: Every partially computable function f: $N \rightarrow N$ over the natural numbers N can be expressed in SQ^+.

Proof: Let $f(n)$ be any partially computable function. Then there is a deterministic Turing machine TM that inputs the string 1^n and outputs the string $1^{f(n)}$ if $f(n)$ is defined, and will never stop otherwise. From this machine, we can construct an SQ^+ expression for $f(n)$ as follows:

We assume that TM has a finite, but ever-growing tape that initially contains only the input in the form of n 1's with one blank cell at each end. After each move of the read-write head, the tape grows by one blank cell at its right end. Note that this tape is equivalent to a one-way infinite tape.

We represent the next-move function δ of TM as a set of *quintuples* [19] of the form:

$[q_i,\ s_j,\ s_k,\ +1,\ q_l]$
or
$[q_i,\ s_j,\ s_k,\ -1,\ q_l]$

The first quintuple signifies that when the machine is in state q_i scanning s_j it will print s_k and then move to the right going into state q_l. The second quintuple is the same, except that the motion is to the left.

We use a 6-tuple $[t, i, d, q, h, c]$ to represent the fact that the content of cell i is d just before step t of the computation, when the machine is in state q, and the read-write head is scanning cell h with content c. We use a set *comp* to collect all the 6-tuples for all tape cells and all computation steps starting from the initial state. Then *comp* is finite if and only if $f(n)$ is defined. Let D be the alphabet of TM, and Q be the set of states of TM.

Now we discuss what should be contained in *comp*.

1. Just before the computation (at step 0), the machine is in state 0, the head is scanning cell 0, which contains a blank (**B**), and cells 1 through n of the tape contain 1's. The last cell (cell n+1) is blank. Therefore, the set

   $$A(n) = \{[0,\ 0,\ \mathbf{B},\ 0,\ 0,\ \mathbf{B}],\ [0,\ n{+}1,\ \mathbf{B},\ 0,\ 0,\ \mathbf{B}]\} \cup \{[0,\ i,\ 1,\ 0,\ 0,\ \mathbf{B}]:\ i = 1,\ ...,\ n\}$$

   must be a subset of *comp*.

2. If *comp* contains a 6-tuple for the cell i at step t, it must also contain a 6-tuple for cell i at step t+1 if t is not the last step. So *comp* must also contain the following set:

   $$B(comp) = \{[t{+}1, i, d, q', h{+}e, c']:\ [t, i, d, q, h, c],\ [t, x, c', q, h, c] \in comp,\ [q, c, s, e, q'] \in \delta \mid i \neq h \textbf{ and } x = h{+}e\} \cup \{[t{+}1, h, s, q', h{+}e, c']:\ [t, x, c', q, h, c] \in comp,\ [q, c, s, e, q'] \in \delta \mid x = h{+}e\}$$

3. Since the tape grows by one blank cell after each step of the computation, *comp* must also contain the following set:

   $$C(n, comp) = \{[t, n{+}t{+}1, \mathbf{B}, q, h, c]:\ [t, i, d, q, h, c] \in comp\}$$

Nothing else will be contained in *comp*.

When the computation stops, the state q of the machine and the symbol d scanned by the head must satisfy the following condition:

$\forall$ [$q1$, $d1$, $d2$, e, $q2$] $\in$ δ | ($q1$ = q $\rightarrow$ $d1$ $\neq$ d)

from which the output F, which is the singleton set containing $f(n)$, can be obtained:

F = #{i: [t, i, x, q, h, d] $\in$ *comp* | x = 1
    and ($\forall$ [$q1$, $d1$, $d2$, e, $q2$] $\in$ δ | ($q1$ = q $\rightarrow$ $d1$ $\neq$ d))}

Thus, the required SQ^+ specification for $f(n)$ is,

(73) **the** *comp*, F:
    comp = $A(n)$ $\cup$ B(*comp*) $\cup$ C(n,*comp*)
    F = #{i: [t, i, 1, q, h, d] $\in$ *comp* |
        $\forall$ [$q1$, $d1$, $d2$, e, $q2$] $\in$ δ | ($q1$ = q $\rightarrow$ $d1$ $\neq$ d)}
  minimizing *comp*, F
  output F

Specification (73) has a finite solution if and only if *TM* halts. ■

In [39], almost the same argument is used to prove that all polynomial time computable relational queries can be expressed in $FO(\geq)$ + **LFP**. The main difference is that the arithmetic operation x + 1 in $FO(\geq)$ + **LFP** is restricted to a range whose size is polynomial to the size of the input domain. It is this restriction that prevents $FO(\geq)$ + **LFP** from simulating a Turing machine that stops in more than a polynomial number of steps.

Since Gurevich [32] has shown that monotonicity is undecidable, we cannot provide an operational semantics for SQ^+. However, we can provide operational semantics for a subset of SQ^+ called $SQ1^+$ in which any expression $\mathbf{LFP}_{\leq,w}(f)$ (respectively, $\mathbf{GFP}_{\leq,w}(f)$) requires that function f must be positive and inflationary (respectively, deflationary) at w (cf. Section 6.3 for how positive and inflationary functions can be defined). We can then define the meaning of expressions $\mathbf{LFP}_{\leq,w}(f)$ and $\mathbf{GFP}_{\leq,w}(f)$ to be the final value of s computed by the following code:

```
s := w
(while f(s) ≠ s)
  s := f(s)
end
```

Corollary 21: Every partially computable function f: $N \rightarrow N$ can be expressed in $SQ1^+$.

Proof: By the rules in Section 6.3 we can recognize that the right-hand-side of both equations in (73) are positive *SQ* expressions monotone in *comp* and F and inflationary at the empty set. ■

With a more sophisticated encoding scheme for the input and output, we can generalize the preceding Corollary to set valued functions:

Corollary 22: Let f: $D_1 \times .. \times D_n \rightarrow D_f$ be a partially computable function, where $D_1, ..., D_n, D_f$ are sets of finite sets of natural numbers. Then $f(x_1, ..., x_n)$ can be expressed in $SQ1^+$ with at most one **LFP** operation. ■

Further generalization to functions over the full range of SQ^+ datatypes is straightforward.

The fact that $SQ1^+$ can specify a Turing machine itself gives rise to another undecidable problem:

Corollary 23: The problem of whether a positive inflationary *SQ* function has a finite fixed point is undecidable in general.

Proof: Otherwise, we could solve the halting problem for Turing machines by translating Turing machines into $SQ1^+$ specifications. ■

Acknowledgement We are grateful to Eric Allender, Barbara Osofsky, Shmuel Sagiv, and Dan Yellin for their valuable comments. Special thanks go to Fritz Henglein for his thorough reading of our paper and for many helpful observations that led to substantial improvements.

Table of Contents

List of Tables