ROPAS Research On Program Analysis System
National Creative Research Initiative Center 1998-2003
Programming Research Laboratory, School of Computer Science & Engineering
Seoul National University

ROPAS MEMO
2004-21
February 9, 2004

A Differential Fixpoint Iteration Method for Static Analysis Specifications

Hyunjun Eo
KAIST [*]

Kwangkeun Yi
Seoul National University [†]

February 9, 2004

Abstract

We present a differential fixpoint iteration method, to be used in static analysis of programs. Computing a static program analysis is done by the fixpoint iterations ("repeat until no-change"), and by "differential" we mean that the method tries to compute only the difference between iterations in order to avoid redundancies. Our method consists of two steps: first we transform the given program analysis function and then we apply our differential fixpoint iteration algorithm to the transformed result. Both steps are fully automatic and does not require that the given analysis function or its input/output lattices should be distributive. Experiments on C, Fortran, and ML programs with realistic analyses show that our method is effective in practice.

1 Problem and Motivation

Computing a static program analysis can always be seen as finding a solution of a set of simultaneous equations, that is, finding a fixpoint x of a function $f : A \to A$ (i.e., $x = f(x)$). The function (we call "analysis function") f must be monotonic ($\forall x \sqsubseteq y. f(x) \sqsubseteq f(y)$) or extensional ($\forall x. x \sqsubseteq f(x)$) over a lattice A. The simultaneous equations (or, the function's body) describe the web of information flows of the program to analyze.

A basic algorithm for computing a fixpoint of function f is to compute the sequence $\{\bot, f(\bot), f^2(\bot), ...\}$ until it stabilizes ($\bot$ is the least element in the lattice A):

```
$x \leftarrow \bot$;
repeat
    $x \leftarrow f(x)$;
until $x$ does not change.
```

Note that because f is monotonic or extensional the sequence $\{\bot, f(\bot), f^2(\bot), ...\}$ is an increasing chain, hence each $f^n(\bot)$ is the join (least upper bound) of the previous result $f^{n-1}(\bot)$ and some increment Δ_n: $f^n(\bot) = f^{n-1}(\bot) \sqcup \Delta_n$.

Thus we can expect computing $f^n(\bot)$ to be accelerated if we reuse the previous result $f^{n-1}(\bot)$ and compute only the increment Δ_n. This efficiency improvement is expected because computing the join $f^{n-1}(\bot) \sqcup \Delta_n$ of the previous result $f^{n-1}(\bot)$ with the difference Δ_n usually costs less than re-applying the whole f to the previous result $f^{n-1}(\bot)$.

Our motivation of looking for such a differential algorithm comes from our project to build a program-analyzer generator named Zoo [7]. Zoo's user (analysis designer) defines a program

[*] Department of Computer Science,
Korea Advanced Institute of Science & Technology,
Email: poisson@ropas.kaist.ac.kr

[†] School of Computer Science & Engineering
Seoul National University
Email: kwang@cse.snu.ac.kr

analysis function in a provided specification language. Zoo then compiles the analysis specification into an executable analyzer whose core computation procedure is the fixpoint iterations. The generated analyzer, given an input program to analyze, derives a set of data-flow equations from the specified analysis function and solves the equations by the fixpoint iterations.

The existing differential fixpoint algorithms [2, 5] are hardly adoptable in our case. They are not general enough; they assume that the analysis functions should be distributive. Because non-distributive functions are frequent in program static analysis, we need to devise a new differential fixpoint algorithm that works for non-distributive functions.

Furthermore, because we have access to the function's definitions we can be more aggressive than existing differential algorithms that assume only functions in extenso. Our method includes a source-level transformation of the analysis function f. The transformed analysis function is then fed into our differential fixpoint iteration algorithm. The transformed function is a differential analysis function f' that satisfies:

1. f' computes the output difference from the input difference Δ. In other words, $f(a \sqcup \Delta) = f(a) \sqcup f'(a, \Delta)$, implying that the join of the previous result $f(a)$ and the difference $f'(a, \Delta)$ makes the current result $f(a \sqcup \Delta)$.

2. computing $f(a) \sqcup f'(a, \Delta)$ costs less than computing $f(a \sqcup \Delta)$. Note that we don't compute $f(a)$ because it is the result from the previous fixpoint iteration.

Section 2 defines an analysis specification language, the language in which the analysis functions are defined. Section 3 presents our transformation method from an analysis specification into a differential one. Section 4 presents our differential fixpoint iteration algorithm. Section 5 shows our experimental results, and Section 6 concludes.

2 Analysis Specification Language

A specification of a program analysis is a set of analysis function definitions over lattices [9]:

analysis	::=	$(\texttt{fun}\ f\ x\ \texttt{=}\ e)^+$	analysis function definition
e	::=	l	constant lattice element
	$\mid$	x	non-function variable
	$\mid$	$f\ e$	analysis function application
	$\mid$	$e \sqcup e \mid e \sqcap e$	join and meet operators
	$\mid$	$(e, e) \mid e.i$	tuple construction and selection
	$\mid$	$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e$	let binding
	$\mid$	$\texttt{if}\ e \sqsubseteq e\ ?\ e\ :\ e$	conditional branch

Semantics of analysis specification language is given in Figure 1. The language is a usual first-order applicative language with the three special operators for lattice elements: join ($\sqcup$), meet ($\sqcap$), and the partial order operator ($\sqsubseteq$). We assume that every analysis function is closed without a free non-function variable and every variable is distinct. The notation "$E + \{x \mapsto v\}$" is for a newly extended environment equal to E except that it maps x to v.

3 Differential Transformation

Figure 2 shows the transformation $\mathcal{T}_\Delta$ of analysis function f which satisfies:

$$f(v \sqcup \Delta v) = f(v) \sqcup \mathcal{T}_\Delta(f)(v, \Delta v).$$

Intuitively, $\mathcal{T}_\Delta(f)(v, \Delta v)$ is the increment of the result induced by the increment in the input argument.

We first transform every function definition $\texttt{fun}\ f\ x\ \texttt{=}\ e$ into the definition of the differential function f_Δ. The differential function f_Δ of f takes the previous argument x and the increment

$$E \vdash l \Downarrow l \qquad \frac{E(x) = v}{E \vdash x \Downarrow v} \qquad \frac{E \vdash e \Downarrow v \quad E + \{x \mapsto v\} \vdash e' \Downarrow v'}{E \vdash f\ e \Downarrow v'}\ (\texttt{fun}\ f\ x\ \texttt{=}\ e' \in \mathit{analysis})$$

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2}{E \vdash e_1 \sqcup e_2 \Downarrow v_1 \sqcup v_2} \qquad \frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2}{E \vdash e_1 \sqcap e_2 \Downarrow v_1 \sqcap v_2}$$

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2}{E \vdash (e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{E \vdash e \Downarrow (v_1, v_2)}{E \vdash e.i \Downarrow v_i}\ (i = 1 \text{ or } 2)$$

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E + \{x \mapsto v_1\} \vdash e_2 \Downarrow v}{E \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow v}$$

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2 \quad v_1 \sqsubseteq v_2 \quad E \vdash e_3 \Downarrow v_3}{E \vdash \texttt{if}\ e_1 \sqsubseteq e_2\ ?\ e_3\ :\ e_4 \Downarrow v_3} \qquad \frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2 \quad v_1 \not\sqsubseteq v_2 \quad E \vdash e_4 \Downarrow v_4}{E \vdash \texttt{if}\ e_1 \sqsubseteq e_2\ ?\ e_3\ :\ e_4 \Downarrow v_4}$$

Figure 1: Semantics of analysis specification language

$$\begin{array}{lcll}
\mathcal{T}_\Delta(\texttt{fun}\ f\ x\ \texttt{=}\ e) & = & \texttt{fun}\ f_\Delta\ (x, \Delta x)\ \texttt{=}\ \mathcal{T}_\Delta(e) & (T1)\\
\mathcal{T}_\Delta(l) & = & \bot & (T2)\\
\mathcal{T}_\Delta(x) & = & \Delta x & (T3)\\
\mathcal{T}_\Delta(f\ e) & = & f_\Delta\ (e, \mathcal{T}_\Delta(e)) & (T4)\\
\mathcal{T}_\Delta(e_1 \sqcup e_2) & = & \mathcal{T}_\Delta(e_1) \sqcup \mathcal{T}_\Delta(e_2) & (T5)\\
\mathcal{T}_\Delta(e_1 \sqcap e_2) & = & \begin{cases} (\mathcal{T}_\Delta(e_1) \sqcap e_2) \sqcup (e_1 \sqcap \mathcal{T}_\Delta(e_2)) \sqcup (\mathcal{T}_\Delta(e_1) \sqcap \mathcal{T}_\Delta(e_2)), \\ \qquad \text{if } e_1 \text{ and } e_2 \text{ are elements of distributive lattice} \\ (e_1 \sqcup \mathcal{T}_\Delta(e_1)) \sqcap (e_2 \sqcup \mathcal{T}_\Delta(e_2)), \quad \text{otherwise} \end{cases} & (T6)\\
\mathcal{T}_\Delta((e_1, e_2)) & = & (\mathcal{T}_\Delta(e_1), \mathcal{T}_\Delta(e_2)) & (T7)\\
\mathcal{T}_\Delta(e.i) & = & \mathcal{T}_\Delta(e).i & (T8)\\
\mathcal{T}_\Delta(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) & = & \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ \texttt{let}\ \Delta x\ \texttt{=}\ \mathcal{T}_\Delta(e_1)\ \texttt{in}\ \mathcal{T}_\Delta(e_2) & (T9)\\
\mathcal{T}_\Delta(\texttt{if}\ e_1 \sqsubseteq e_2\ ?\ e_3\ :\ e_4) & = & \texttt{if}\ (e_1 \sqcup \mathcal{T}_\Delta(e_1)) \sqsubseteq (e_2 \sqcup \mathcal{T}_\Delta(e_2)) & \\
 & & \quad ?\ (\texttt{if}\ e_1 \sqsubseteq e_2\ ?\ \mathcal{T}_\Delta(e_3)\ :\ e_3 \sqcup \mathcal{T}_\Delta(e_3)) & \\
 & & \quad :\ (\texttt{if}\ e_1 \sqsubseteq e_2\ ?\ e_4 \sqcup \mathcal{T}_\Delta(e_4)\ :\ \mathcal{T}_\Delta(e_4)) & (T10)
\end{array}$$

Figure 2: Differential transformation $\mathcal{T}_\Delta$

of the argument Δx, and compute the increment of body e ($T1$). From these differential function definitions, we transform function application $f\ e$ into the application of differential function f_Δ to the both of the previous argument e and the difference $\mathcal{T}_\Delta(e)$ ($T4$). We transform `let`-bindings in the same way ($T9$). We transform a constant l to $\bot$ because it remains unchanged throughout the fixpoint iteration ($T2$).

Transformation of $e_1 \sqcup e_2$, (e_1, e_2), and $e.i$ is simple. Because the $\sqcup$-operation is distributive over the operations, i.e., $(e_1 \sqcup \Delta_1) \sqcup (e_2 \sqcup \Delta_2) = (e_1 \sqcup e_2) \sqcup (\Delta_1 \sqcup \Delta_2)$, $(e_1 \sqcup \Delta_1, e_2 \sqcup \Delta_2) = (e_1, e_2) \sqcup (\Delta_1, \Delta_2)$, and $(e \sqcup \Delta).i = e.i \sqcup \Delta.i$,, we just replace each subexpression into a differential one: ($T5$), ($T7$), and ($T8$).

Transformation ($T6$) of $e_1 \sqcap e_2$ has two choices, depending on whether or not the meet operator is over distributive lattices. Over distributive lattices, we can transform $e_1 \sqcap e_2$ by the distributive law: $\mathcal{T}_\Delta(e_1 \sqcap e_2) = (\mathcal{T}_\Delta(e_1) \sqcap e_2) \sqcup (e_1 \sqcap \mathcal{T}_\Delta(e_2)) \sqcup (\mathcal{T}_\Delta(e_1) \sqcap \mathcal{T}_\Delta(e_2))$. Over non-distributive lattices, we should be conservative: $\mathcal{T}_\Delta(e_1 \sqcap e_2) = (e_1 \sqcup \mathcal{T}_\Delta(e_1)) \sqcap (e_2 \sqcup \mathcal{T}_\Delta(e_2))$.

For `if`-branch ($T10$), when the conditional $e_1 \sqsubseteq e_2$'s values remain the same, the result of `if`-branch is only the difference of either the true or the false branch: $\mathcal{T}_\Delta(e_3)$ or $\mathcal{T}_\Delta(e_4)$. However, if the results of the conditional expression changes, we have to join the previous result with the current difference.

Theorem 1 shows that our transformation is correct: it satisfies $f(a \sqcup b) = f(a) \sqcup \mathcal{T}_\Delta(f)(a, b)$.

We naturally extend the transformation $\mathcal{T}_\Delta$ for environment E such as:

$$\begin{array}{lcl}\mathcal{T}_\Delta(\{x \mapsto v\}) & = & \{\Delta x \mapsto v\} \\ \mathcal{T}_\Delta(E + E') & = & \mathcal{T}_\Delta(E) + \mathcal{T}_\Delta(E'), \text{where } E + E' = \{x \mapsto v | x \notin \mathit{dom}(E'), x \mapsto v \in E\} \cup E'.\end{array}$$

To prove the theorem, we need the following lemma.

Lemma 1 *For all environments E and E', and expression e, if $E \vdash e \Downarrow v$ and $E \sqcup E' \vdash e \Downarrow v'$ then $E + \mathcal{T}_\Delta(E') \vdash \mathcal{T}_\Delta(e) \Downarrow \Delta v$ and $v' = v \sqcup \Delta v$.*

Proof. We proceed by structural induction on e. We only show the case of function application; other cases are similarly proven.

- For function application $f\ e$, we have to show that $E \vdash f\ e \Downarrow v$ and $E \sqcup E' \vdash f\ e \Downarrow v'$ implies $E + \mathcal{T}_\Delta(E') \vdash \mathcal{T}_\Delta(f\ e) \Downarrow \Delta v$ and $v' = v \sqcup \Delta v$. By the semantics of function application, $E \sqcup E' \vdash f\ e \Downarrow v'$ implies

$$E \sqcup E' \vdash e \Downarrow v'_e \quad (1)$$
$$(E \sqcup E') + \{x \mapsto v'_e\} \vdash e' \Downarrow v', \quad (2)$$

  where "`fun` $f\ x$ `=` e'" is in the analysis specification. Similarly, $E \vdash f\ e \Downarrow v$ implies

$$E \vdash e \Downarrow v_e \quad (3)$$
$$E + \{x \mapsto v_e\} \vdash e' \Downarrow v. \quad (4)$$

  Because $\mathcal{T}_\Delta($`fun` $f\ x$ `=` $e') =$ `fun` $f_\Delta\ (x, \Delta x)$ `=` $\mathcal{T}_\Delta(e')$, we have to show that

$$E + \mathcal{T}_\Delta(E') \vdash f_\Delta(e, \mathcal{T}_\Delta(e)) \Downarrow \Delta v.$$

  By induction hypothesis, (1) and (3) implies

$$E + \mathcal{T}_\Delta(E') \vdash \mathcal{T}_\Delta(e) \Downarrow \Delta v_e \quad \text{and} \quad v'_e = v_e \sqcup \Delta v_e. \quad (5)$$

  Because $(E \sqcup E') + \{x \mapsto v'_e\} = (E \sqcup E') + \{x \mapsto v_e \sqcup \Delta v_e\} = (E + \{x \mapsto v_e\}) \sqcup (E' + \{x \mapsto \Delta v_e\})$, (2) implies

$$(E + \{x \mapsto v_e\}) \sqcup (E' + \{x \mapsto \Delta v_e\}) \vdash e' \Downarrow v'. \quad (6)$$

  Then, by induction hypothesis, (6) and (4) implies

$$E + \{x \mapsto v_e\} + \mathcal{T}_\Delta(E' + \{x \mapsto \Delta v_e\}) \vdash \mathcal{T}_\Delta(e') \Downarrow \Delta v \quad (7)$$

  and $v' = v \sqcup \Delta v$. By definition of $\mathcal{T}_\Delta$, (7) is

$$E + \{x \mapsto v_e\} + \mathcal{T}_\Delta(E') + \{\Delta x \mapsto \Delta v_e\} \vdash \mathcal{T}_\Delta(e') \Downarrow \Delta v$$

  and in turn, because $\mathit{dom}\mathcal{T}_\Delta(E') \not\ni x$,

$$E + \mathcal{T}_\Delta(E') + \{x \mapsto v_e\} + \{\Delta x \mapsto \Delta v_e\} \vdash \mathcal{T}_\Delta(e') \Downarrow \Delta v. \quad (8)$$

  Because Δx does not occur free in analysis expression e, (3) implies

$$E + \mathcal{T}_\Delta(E') \vdash e \Downarrow v_e. \quad (9)$$

  By the semantics of function application, (9), (5), and (8) implies

$$E + \mathcal{T}_\Delta(E') \vdash f_\Delta(e, \mathcal{T}_\Delta(e)) \Downarrow \Delta v. \qquad \square$$

analysis expr. e	cost C_e of computing e	cost $C_{e'}$ of computing the differential version $e' = \mathcal{T}_\Delta(e)$
l	$O(1)$	$O(1)$
x	$O(1)$	$O(1)$
$f\ e_1$	$C_{e_1} + C_f$	$C_{e_1} + C_{e'_1} + C_{f'}$
$e_1 \sqcup e_2$	$C_{e_1} + C_{e_2} + C_\sqcup$	$C_{e'_1} + C_{e'_2} + C_\sqcup$
$e_1 \sqcap e_2$ *	$C_{e_1} + C_{e_2} + C_\sqcap$	$C_{e_1} + C_{e'_1} + C_{e_2} + C_{e'_2} + C_\sqcap + C_\sqcup$
$e_1 \sqcap e_2$ **	$C_{e_1} + C_{e_2} + C_\sqcap$	$C_{e_1} + C_{e'_1} + C_{e_2} + C_{e'_2} + C_\sqcap + C_\sqcup$
(e_1, e_2)	$C_{e_1} + C_{e_2}$	$C_{e'_1} + C_{e'_2}$
$e_1.i$	C_{e_1}	$C_{e'_1}$
`let` x `=` e_1 `in` e_2	$C_{e_1} + C_{e_2}$	$C_{e_1} + C_{e'_1} + C_{e'_2}$
`if` $e_1 \sqsubseteq e_2\ ?\ e_3\ :\ e_4$	$C_{e_1} + C_{e_2} + C_\sqsubseteq + \max(C_{e_3}, C_{e_4})$	$C_{e_1} + C_{e'_1} + C_{e_2} + C_{e'_2} + C_\sqsubseteq + C_\sqcup + \max(C_{e'_3}, C_{e'_4})$

* $\sqcap$-operation over distributive lattices
** $\sqcap$-operation over non-distributive lattices

Figure 3: Dominant time-complexity terms form computing an analysis expression e versus its differential version e'

Theorem 1 *For all analysis function f defined by "`fun` f x `=` e" in the analysis specification and for all lattice elements a and b, $f(a \sqcup b) = f(a) \sqcup \mathcal{T}_\Delta(f)(a, b)$.*

Proof. Let v', v, and Δv be $\vdash f(a \sqcup b) \Downarrow v'$, $\vdash f(a) \Downarrow v$, and $\vdash \mathcal{T}_\Delta(f)(a, b) \Downarrow \Delta v$, respectively. Then we have to show that $v' = v \sqcup \Delta v$. Because f is defined in the analysis specification as "`fun` f x `=` e," we can get following judgments by the semantics of function definition and function application:

$$\{x \mapsto a \sqcup b\} \vdash e \Downarrow v',\ \{x \mapsto a\} \vdash e \Downarrow v,\ \text{and } \{x \mapsto a, \Delta x \mapsto b\} \vdash \mathcal{T}_\Delta(e) \Downarrow \Delta v.$$

Let $E = \{x \mapsto a\}$ and $E' = \{x \mapsto b\}$. Then $v' = v \sqcup \Delta v$ by Lemma 1. Thus $f(a \sqcup b) = f(a) \sqcup \mathcal{T}_\Delta(f)(a, b)$. □

The time complexity, in the big O notation, for computing the differential expression remains the same as that for computing the original expression:

Theorem 2 *For all analysis expression e and its transformed expression $e' = \mathcal{T}_\Delta(e)$, let C_e and $C_{e'}$ respectively be the time complexities in the big O notation of computing e and e'. Then $C_{e'} = C_e$.*

Proof. It is straightforward by structural induction on e. e_i. Figure 3 enumerates, in the big O notation, dominant time-complexity terms for computing differential version e' versus those for computing original expression e. The theorem holds because 1) complexity terms for differential e' is always a linear combination of those for original e, 2) $C_\sqcup = C_\sqcap = C_\sqsubseteq$, and 3) by the induction hypothesis being $C_{e_i} = C_{e'_i}$ for every sub-expression e_i. □

Note that though computing the differential expression e' has to apply more join or meet operations than computing the original e (Figure 3), the computation cost can actually be reduced because the arguments to the operators are reduced differential ones. Our experiments in Section 5 back up this claim.

4 Differential Fixpoint Iteration Algorithm

Algorithm $\mathcal{D}$ in Figure 4 is a basic algorithm for the differential fixpoint iteration. We first initialize x and Δx respectively by $\bot$ and $f(\bot)$ $(D1)$. Δx should be initialized to $f(\bot)$ because it is the initial difference:

$$f^n(\bot) \quad = \quad f^{n-1}(\bot) \sqcup \Delta_{n-1} = \cdots = f(\bot) \sqcup \Delta_1 \sqcup \Delta_2 \sqcup ... \sqcup \Delta_{n-1}.$$

```
(D1)    $x \leftarrow \bot$; $\Delta x \leftarrow f(\bot)$;
        repeat
(D2)        $x_p \leftarrow x$; $\Delta x_p \leftarrow \Delta x$;
(D3)        $x \leftarrow x_p \sqcup \Delta x_p$;
(D4)        $\Delta x \leftarrow \mathcal{T}_\Delta(f)(x_p, \Delta x_p) \setminus x$;
        until $x = x_p$;
```

Figure 4: Differential fixpoint iteration algorithm $\mathcal{D}$

Then we repeat computing x and Δx until the current result x and the previous result x_p are the same, that is, it reaches a fixpoint. Line ($D2$) records the previous x and Δx in x_p and Δx_p, respectively. Line ($D3$) computes the current result x and line ($D4$) computes the current difference Δx. In order to get the exact difference, we subtract x from the $\mathcal{T}_\Delta(f)(x_p, \Delta x_p)$. The difference $a \setminus b$ of two lattice elements a and b is required to satisfy:

$$(a \setminus b) \sqcup b = a \sqcup b \quad \text{and} \quad (a \setminus b) \sqsubseteq a. \tag{10}$$

In worst case, $a \setminus b = a$, and in best case $(a \setminus b) \sqcap b = \bot$. For example, when domain L is a powerset lattice, we use set-minus operation which satisfies $(a \setminus b) \sqcap b = \bot$. For a flat lattice, $a \setminus b = \bot$ if $a \sqsubseteq b$, and $a \setminus b = a$ otherwise. This subtraction is necessary for reducing the difference as much as possible. The cost of $\setminus$-operation is the same as the cost of $\sqcup$-operation and it is usually less than the cost of $\mathcal{T}_\Delta(f)(x_p, \Delta x_p)$.

Theorem 3 *For all analysis function f, if f is monotonic or extensional, algorithm $\mathcal{D}$ computes a fixpoint of f.*

Proof. Let $x_0 = f(\bot)$ and $\Delta x_0 = \bot$ from ($D1$). Let x_i and Δx_i be respectively the values of x and Δx of ith iteration such that

$$x_i = x_{i-1} \sqcup \Delta x_{i-1} \qquad \text{from } (D2) \text{ and } (D3) \tag{11}$$

$$\Delta x_i = \mathcal{T}_\Delta(f)(x_{i-1}, \Delta x_{i-1}) \setminus x_i \qquad \text{from } (D2) \text{ and } (D4). \tag{12}$$

Then we can show by induction on i that $x_i = f^i(\bot)$ for all $i \geq 1$.

- (Base case) $i = 1$: $\quad x_1 = x_0 \sqcup \Delta x_0 = \bot \sqcup f(\bot) = f(\bot)$.
- (Induction step) We assume the theorem holds for $i \leq n$. Then for the case of $i = n+1$:

$$\begin{array}{rcll}
x_{n+1} & = & x_n \sqcup \Delta x_n & \text{by (11)} \\
& = & x_n \sqcup (\mathcal{T}_\Delta(f)(x_{n-1}, \Delta x_{n-1}) \setminus x_n) & \text{by (12)} \\
& = & x_n \sqcup \mathcal{T}_\Delta(f)(x_{n-1}, \Delta x_{n-1}) & \text{by (10)} \\
& = & f^n(\bot) \sqcup \mathcal{T}_\Delta(f)(f^{n-1}(\bot), \Delta x_{n-1}) & \text{by induction hypothesis} \\
& = & f(f^{n-1}(\bot)) \sqcup \mathcal{T}_\Delta(f)(f^{n-1}(\bot), \Delta x_{n-1}) & \text{by definition} \\
& = & f(f^{n-1}(\bot) \sqcup \Delta x_{n-1}) & \text{by Theorem 1} \\
& = & f(x_{n-1} \sqcup \Delta x_{n-1}) & \text{by induction hypothesis} \\
& = & f(x_n) & \text{by (11)} \\
& = & f(f^n(\bot)) & \text{by induction hypothesis} \\
& = & f^{n+1}(\bot). &
\end{array}$$

Because function f is monotonic or extensional, algorithm $\mathcal{D}$ computes a fixpoint of function f. In case that f is monotonic, algorithm $\mathcal{D}$ computes the least fixpoint of function f. □

Corollary 1 *The result of differential algorithm $\mathcal{D}$ is exactly the same as the result of non-differential algorithm.*

Proof. For each iteration i, both algorithms compute the same result $f^i(\bot)$. □

5 Experiments

We implemented a prototype of our transformation and differential algorithm inside System Z1 [9] (a predecessor of our planned Zoo). We implemented a work-list version [2, 5] of the differential fixpoint iterations and used the well-known optimization techniques in realistic implementations such as hash-consing and memoization.

Our experiments aim to make sure that first, our method works for higher-order analysis[1] and second, it scales up. For higher-order analysis, we experimented the exception analysis [8] of ML programs. For scale-up testing, we used the constant propagation [6] and alias (cp-alias) analysis of C and Fortran programs. We chose cp-alias analysis for the scale-up testing because the analysis domain (powerset lattice) allows a large range of lattice height, hence we can examine the analysis cost for each lattice height. Both the exception analysis and the cp-alias analysis are not distributive.

Figure 5 shows the effectiveness of our differential method for the higher-order analysis (exception analysis). It shows that our method is effective for an analysis that requires control flow analysis of higher-order language. For each benchmark program, differential method saves about 28–53% of execution time.

Figure 6 shows our method's scalability for cp-alias analysis. It shows that our method keeps reducing the analysis time such that the analysis cost is kept almost linear to the height of the powerset lattice. Because the time complexity of fixpoint iteration is $O(h \times n) \times C_f$, where h is the height of the analysis lattice, n the program size, and C_f the cost for applying analysis function f to the previous result, Figure 6 shows that the cost of applying the differential analysis function f_Δ takes almost constant time.

6 Discussion

Combining our differential transformation of the input analysis function and our differential fixpoint iterations is proven correct and its experiments in realistic settings show it's a promising approach. Though the combination method is intended for use inside our automatic program analyzer generator, the ideas can also be used by program analysis implementers in manually tuning their fixpoint procedures. Note also that the widely-used widening and narrowing operations [3] that accelerate the fixpoint iterations do not interfere with our method.

Ahn and Kwon's work [1] differs from ours in two points. Their method is less differential than ours; we compute and reuse the difference between current and previous results, whereas they just reuse the previous result. By computing the difference, we can further reduce the increment Δ. Second, they do not transform the analysis functions; their algorithm interprets the analysis functions in a differential way, while we "compile" the functions hence can apply several a priori optimizations (e.g. dead-code elimination and common subexpression elimination) as reported in [4].

References

[1] J. Ahn, Y. J. Kwon, A differential fixpoint evaluation framework for non-distributive systems, in: Proceedings of the First Asian Symposium on Programming Languages and Systems, in: Lecture Notes in Computer Science, vol. 2895, Springer-Verlag, Beijing, 2003, pp. 159–175.

[2] J. Cai, R. Paige, Program derivation by fixed point computation, Science of Computer Programming 11(3) (1989) 197–261.

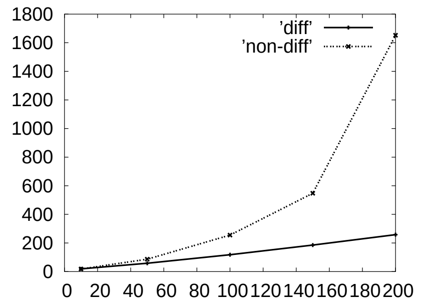
[1] A static analysis is called "higher-order" if it can analyze higher-order programs that compute functions as first-class objects like in ML or Scheme.

program	exprs[a]	non-diff. (secs[b])	diff. (secs[c])	speed up[d]
`lexgen.sml`	14527	281.60	135.54	52%
`mlyacc.sml`	74198	1733.21	815.86	53%
`libkin.sml`	15837	115.70	76.50	51%
`libkin2.sml`	15837	89.57	64.18	28%

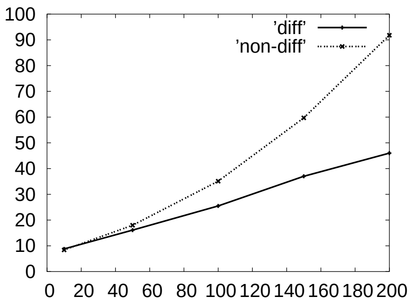
[a]the number of expressions in a program
[b]CPU execution time (seconds) for non-differential method. We experimented on a Sun Enterprise 450 server with dual Ultra Sparc 400MHz CPUs and 2GBs memory.
[c]CPU execution time (seconds) for differential method
[d]speed up = (c − d)/c

Figure 5: Our differential algorithm is effective

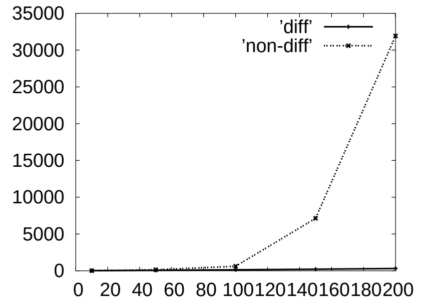
height[a]	`amoeba.f`		`gauss.f`		`TIS.f`		`wator.c`	
	non-diff	diff	non-diff	diff	non-diff	diff	non-diff	diff
10	18.3	19.1	8.4	8.8	16.3	16.3	17.5	23.7
50	86.3	58.1	18.0	16.1	141.9	70.0	97.7	104.4
100	254.9	118.0	35.1	25.5	619.4	140.8	291.8	202.4
150	547.6	185.2	59.7	37.0	7131.6	212.2	1118.7	308.1
200	1652.0	258.2	91.8	46.0	31926.0	306.2	3476.3	422.3



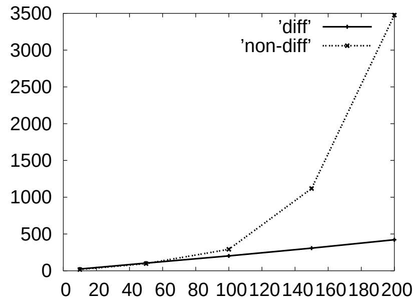
`amoeba.f` (6062[b])



`TIS.f` (6028)



`gauss.f` (4710)



`wator.c` (3467)

[a]the height of analysis domain (lattice)
[b]the number of expressions in a program

Figure 6: Our differential algorithm scales up

[3] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977, pp. 238-252.

[4] H. Eo, K. Yi, An improved differential fixpoint iteration method for program analysis, in: Proceedings of the Third Asian Workshop on Programming Languages and Systems,

Shanghai, 2002.

[5] C. Fecht, H. Seidl, Propagating differences: an efficient new fixpoint algorithm for distributive constraint systems, in: Proceedings of European Symposium on Programming, in: Lecture Notes in Computer Science, vol. 1381, Springer-Verlag, Lisbon, 1998, pp. 90-104.

[6] M. N. Wegman, F. K. Zadeck, Constant propagation with conditional branches, ACM Transactions on Programming Languages and Systems 13(2) (1991) 181–210.

[7] K. Yi, Program Analysis System Zoo, Research On Programming Languages and Systems, Seoul National University, http://ropas.snu.ac.kr/zoo

[8] K. Yi, An abstract interpretation for estimating uncaught exception in Standard ML programs, Science of Computer Programming 31(1) (1998) 147–173.

[9] K. Yi, W. L. Harrison III, Automatic generation and management of interprocedural program analyses, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, 1993, pp. 246–259.