Research On Program Analysis System
National Creative Research Initiative Center
ROPAS Korea Advanced Institute of Science and Technology

ROPAS MEMO
2001-15
November 19, 2001

Static Monotonicity Analysis for λ-Definable Functions over Lattices *†

Andrzej Murawski
Computing Laboratory
Oxford University
andrzej@comlab.ox.ac.uk

Kwangkeun Yi
Research On Program Analysis System‡
KAIST §
kwang@cs.kaist.ac.kr

November 19, 2001

Abstract

We employ static analysis to examine monotonicity of functions defined over lattices in a λ-calculus augmented with constants, branching, meets, joins and recursive definitions. The need for such a verification procedure has recently arisen in our work with a static analyzer generator called Zoo, in which the specification of static analysis (input to Zoo) consists of finite-height lattice definitions and function definitions over the lattices. Once monotonicity of the functions is ascertained, the generated analyzer is guaranteed to terminate.

1 Motivation

We are currently involved in a project to build a program-analyzer generator (named "Zoo" [Yi01a, Yi01b]). One of the program analysis frameworks that Zoo supports is abstract interpretation [CC77, CC92]. Its user (analysis designer) defines an abstract interpreter in a specification language (named "Rabbit"). Zoo then compiles the input Rabbit program into an executable analyzer which, given an input program to analyze, derives a set of data-flow equations and solves them by fixpoint iterations.

Zoo, as of now, is less discerning than desirable; it does not check whether the user-specified abstract interpreter defines a correct and terminating analysis. It blindly generates an executable program without verifying that the input specification qualifies for static analysis. Assuring correctness and termination of the specified abstract interpreter is the responsibility of the designer (Zoo's user).

To overcome these shortcomings, we have designed a static analysis method by which Zoo can check the monotonicity of the input abstract interpreters. An abstract interpreter consists of lattice definitions and definitions of functions over the lattices. Once it is known that the functions are monotonic, the generated analyzers

*This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

†This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

‡http://ropas.kaist.ac.kr

§Korea Advanced Institute of Science & Technology are guaranteed to terminate (because Zoo allows only finite-height lattices). By using the analysis, Zoo can statically estimate the monotonicity of the input functions and consequently reject analyzers whose specification is possibly not monotonic.

Existing results [Vor00, DGL+99, GGLR98, Sch96] on monotonicity verification in learning theory have turned out hardly adoptable in our case. They are restricted to boolean lattices and concern functions $\{0,1\}^n \to \{0,1\}$. Though finite distributive lattices can be embedded in a product of the boolean lattices [Rut65], Zoo also supports non-distributive lattices which are prevalent in static analysis. The above-mentioned algorithms are probabilistic, and so are allowed to err with some small probability. In our generalized case, finding a tight bound on this probability of mistakes seems a formidable job. Besides, only functions in extenso seem to have been studied thus far, whereas we also have access to the definitions. This makes the problem amenable to static analysis. Furthermore, what if conventional static analysis can reliably ensure the monotonicity with a reasonable accuracy? This is the approach we took and we present the outcome in this paper.

2 Setting

Let L_1 and L_2 be lattices. A function $f : L_1 \longrightarrow L_2$ is monotonic (respectively anti-monotonic) if and only if for all $x \leq y$, we have $f(x) \leq f(y)$ (respectively $f(y) \leq f(x)$). If a function is both monotonic and anti-monotonic, it is constant. Analogously, for functions of many arguments, we can define monotonicity and anti-monotonicity with respect to the ith argument.

Our goal is to design a static procedure that can certify whether a function between two lattices is monotonic or not. The source language is Rabbit [Yi01a], the input specification language of the Zoo system. For brevity of presentation, we consider the following core as the source language:

e	$::=$	c	constant (lattice point)
	$\mid$	x	variable
	$\mid$	$\lambda x.e$	function
	$\mid$	$\mathit{fix}\ f\ e$	recursive definition
	$\mid$	$e\ e$	application
	$\mid$	$e \sqcup e$	join operation
	$\mid$	$e \sqcap e$	meet operation
	$\mid$	$\mathit{if}\ e \sqsubseteq e\ \mathit{then}\ e\ \mathit{else}\ e$	branching

Values in this language are either lattice elements or functions over lattices. c is a constant expression denoting a lattice element. The *if* expression branches, as usual, depending on whether the conditional partial-order relation holds or not. In the actual Rabbit language [Yi01a] one can also compute elements in lattices of various kinds: product lattices, powerset lattices, function lattices, and lattices with user-defined orders.

We write $e(x_1, \cdots, x_n)$ if $\{x_1, \cdots, x_n\}$ are the free variables of e. We also write $e(c_1, \cdots, c_n)$ when a constant c_i is substituted for each x_i in e.

3 Monotonicity Checking by an Effect Type System

Given an expression e of the core language, our monotonicity check will determine conservatively for each $1 \leq i \leq n$ whether the operation

$$(x_1, \cdots, x_n) \mapsto e(x_1, \cdots, x_n)$$

is monotonic, anti-monotonic, or constant with respect to the ith argument. This monotonicity behavior will be summarized in a table. For example, the expression $x \sqcup c$ defines $\{x \mapsto \text{monotonic}, \textit{else} \mapsto \text{constant}\}$: monotonic for the free variable x, constant for other variables. As another example take *if* $x \sqsubseteq c$ *then* $\top$ *else* $\bot$. Here the monotonicity is captured by $\{x \mapsto \text{anti-monotonic}, \textit{else} \mapsto \text{constant}\}$, because the values change from $\top$ to $\bot$ (decreasing) as x increases.

We present the procedure as an effect-type inference system with typing judgments of the form

$$\Gamma \vdash e : \tau, me.$$

The judgments should be read as "under type environment Γ, expression e has type τ and monotonicity behavior me". The monotonicity behavior is a function

$$me \quad \in \quad ME = \textit{Var} \to M$$

from the set of variables to the set M of *monotonicity tokens*

$$m \quad \in \quad M = \{0, +, -, \top\}.$$

We normally write me in table form $\{\cdots\}$. Monotonicity tokens have the following meaning:

$$\begin{array}{lcl}
[\![0]\!] & = & \{f \mid x \sqsubseteq y \text{ implies } f(x) = f(y) \text{ if } f(x), f(y) \text{ terminate}\} \\
[\![+]\!] & = & \{f \mid x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y) \text{ if } f(x), f(y) \text{ terminate}\} \\
[\![-]\!] & = & \{f \mid x \sqsubseteq y \text{ implies } f(y) \sqsubseteq f(x) \text{ if } f(x), f(y) \text{ terminate}\} \\
[\![\top]\!] & = & \text{all functions}
\end{array}$$

and hence they form a diamond-shaped lattice:

$$0 \sqsubseteq + \sqsubseteq \top, \quad 0 \sqsubseteq - \sqsubseteq \top.$$

The order on M can be extended to ME in a point-wise fashion:

$$me_1 \sqsubseteq me_2 \ \textit{iff} \ \forall x \in \textit{Var}.me_1(x) \sqsubseteq me_2(x).$$

Types are either ground types ι denoting lattices[1] or function types $\tau \xrightarrow{me} \tau$ with latent monotonicity behavior (effect) me:

$$\textit{Type} \qquad \tau \quad ::= \quad \iota \ \mid \ \tau \xrightarrow{me} \tau$$

The monotonicity behavior me will be described by *monotonicity expressions* which are generated as follows:

$$\begin{array}{lcl}
me & ::= & \bar{0} \mid \bar{+} \mid \bar{-} \mid \{x \mapsto m\} \\
 & \mid & me[m/x] \mid @\ me\ me\ me \\
 & \mid & \textbf{if}\ me\ me\ me\ me\ \phi \mid \textbf{ifc}\ me\ me\ \phi
\end{array}$$

[1] Our results are independent of the choice of lattices denoted by ground types.

$\bar{0}$, $\bar{+}$ and $\bar{-}$ denote respectively all-constant, all-monotonic, and all-anti-monotonic behavior. $\{x \mapsto +\}$ means monotonic in x and constant for others, i.e. the induced function is independent of variables other than x. Similarly, for $\{x \mapsto -\}$ and $\{x \mapsto \top\}$. $me[m/x]$ denotes a table whose entry is m for x, but which is otherwise the same as me. In what follows we define the operators @, **if**, and **ifc** as we introduce the typing rules.

A constant expression remains constant for any variable, hence $\bar{0}$:

$$\frac{}{\Gamma \vdash c : \iota, \bar{0}} \qquad \text{(CON)}$$

The identity function is monotonic, so a variable should be declared as monotonic with respect to itself and constant otherwise:

$$\frac{}{\Gamma \vdash x : \Gamma(x), \{x \mapsto +\}} \qquad \text{(VAR)}$$

The monotonicity of the join operation is compositional:

$$\frac{\Gamma \vdash e_1 : \tau, me_1 \quad \Gamma \vdash e_2 : \tau, me_2}{\Gamma \vdash e_1 \sqcup e_2 : \tau, me_1 \sqcup me_2} \qquad \text{(LUB)}$$

Note that this means that the monotonicity of the two subexpressions is reflected by the monotonicity of the whole term. For example, if e_1 is monotonic and e_2 is anti-monotonic, the result is unknown ($\top$). The same applies to the meet operation. The monotonicity of the expression $e_1 \sqcap e_2$ would also correspond to $me_1 \sqcup me_2$.

The rule for lambda expressions is the same as in any standard effect-type system. The monotonicity behavior of the body is used to annotate the function type as a latent effect. Note that this potential effect can be weaker than that of the body ($me' \sqsubseteq me$). This relaxation makes the rule safely less restrictive; without it we would have to reject programs in which two functions of varying monotonicity are called in the same application. Lastly, the behavior of a lambda expression is identical to that of its body, except that the new function is independent of the freshly bound parameter:

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2, me' \quad me' \sqsubseteq me}{\Gamma \vdash \lambda x.e : \tau_1 \stackrel{me}{\to} \tau_2, me[0/x]} \qquad \text{(LAM)}$$

The rule for recursion requires that the body and the name have the same effect types:

$$\frac{\Gamma + f : \tau \vdash e : \tau, me}{\Gamma \vdash \mathit{fix}\ f\ e : \tau, me[0/f]} \qquad \text{(FIX)}$$

For application we introduce a special operator @:

$$\frac{\Gamma \vdash e_1 : \tau_1 \stackrel{me_1}{\to} \tau_2, me_2 \quad \Gamma \vdash e_2 : \tau_1, me3.}{\Gamma \vdash e_1\ e_2 : \tau_2, @\ me_1\ me_2\ me_3} \qquad \text{(APP)}$$

Although @ could just be defined as taking joins, we can do better for an increased accuracy. First, suppose the function to be called is fixed. When both its body and the argument exhibit the same monotonicity (both increasing or both decreasing), the result of the application will be monotonic (increasing). When one is monotonic (increasing) and the other is anti-monotonic (decreasing), then the application is anti-monotonic (decreasing). When one of the two (body or argument) remains constant, the result is constant. Now, consider the situation in which the function itself is changing, for example, monotonically. Then the application is monotonic only when the argument and the body combined are monotonic. The behavior is unpredictable if the argument and the body combined are anti-monotonic. All these cases (and the remaining ones) are accounted for by:

$$@\ me_{\text{body}}\ me_{\text{ftn}}\ me_{\text{arg}} = (me_{\text{body}} \otimes\ me_{\text{arg}})\ \sqcup\ me_{\text{ftn}}$$

where $me_1 \otimes me_2$ is the pointwise (commutative and monotonic) "multiplication of signs": $+ \otimes + = +$, $- \otimes - = +$, $+ \otimes - = -$ and $0 \otimes \text{any} = 0$.

The case of the conditional expression is quite involved because of the **if** operator:

$$\frac{\Gamma \vdash e_1 : \tau', me_1 \quad \Gamma \vdash e_2 : \tau', me_2 \quad \Gamma \vdash e_3 : \tau, me_3 \quad \Gamma \vdash e_4 : \tau, me_4}{\Gamma \vdash \mathit{if}\ e_1 \sqsubseteq e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4 : \tau, \dot{\mathbf{if}}\ me_1\ me_2\ me_3\ me_4\ \Phi} \quad \text{(IF)}$$

Before we present a definition of $\dot{\mathbf{if}}$, let us note that it is wrong to join the monotonicity behaviors of the two branches. For example, *if* $x \sqsubseteq c$ *then* $\top$ *else* $\bot$ has constant branches but it decreases (switches from $\top$ to $\bot$) as x increases. Thus, we have to examine whether the monotonicity behavior is preserved at the point of the switch and thereafter. We need to know two details: (1) in which direction (from true to false or the reverse) the *if*-condition changes, and (2) whether the consequent change of branches preserves the monotonicity. Our point-wise definition of $\dot{\mathbf{if}}$:

$$\dot{\mathbf{if}}\ me_1\ me_2\ me_3\ me_4\ \Phi = \{x \mapsto \dot{\mathbf{if}}\ me_1(x)\ me_2(x)\ me_3(x)\ me_4(x)\ \Phi \mid x \in \mathit{Var}\}$$

is based on a conservative approximation of the two pieces of information. Assuming, for simplicity, that there exists only one free variable that can occur in each e_i, the four representative cases in the definition of $\dot{\mathbf{if}}$ are as follows:

$me_1(x)$	$me_2(x)$	$me_3(x)$	$me_4(x)$	Φ	$\dot{\mathbf{if}}\ me_1(x) \cdots me_4(x)\ \Phi$
$-$	$+$	$+$	$+$	$e_3(\bot) \sqsupseteq e_4(\top)$	$+$
$-$	$+$	$-$	$-$	$e_3(\bot) \sqsubseteq e_4(\top)$	$-$
$+$	$-$	$+$	$+$	$e_3(\top) \sqsubseteq e_4(\bot)$	$+$
$+$	$-$	$-$	$-$	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$

For example, the first row captures the case when the boolean value of $e_1 \sqsubseteq e_2$ switches from false to true (because e_1 is decreasing and e_2 is increasing). Thus, if the maximal value in the 'false' branch (i.e. $e_4(\top)$, because e_4 is monotonic) does not exceed the minimal value in the 'true' branch ($e_3(\bot)$), we can conclude that the whole *if*-expression is monotonic. In general, for expressions with several variables, the extrema are calculated on the basis of the monotonicity table, e.g. if $e_3(x_1, x_2, x_3)$ defines $\{x_1 \mapsto +, x_2 \mapsto -, x_3 \mapsto +\}$, its smallest value will be $e_3(\bot, \top, \bot)$.

The Φ parameter ensures that monotonicity will be preserved at the switching point. The monotonicity tokens for e_1 and e_2 give a conservative estimate of the direction of the switch. The four cases we have distinguished handle all the posibilities in which monotonicity of the aggregate expression is predictable, provided the participating functions are monotonic or anti-monotonic. There are a few more cases taking constant functions into account. The required results for those are easily derivable from the above table, e.g. (Appendix A has the full definition.)

$me_1(x)$	$me_2(x)$	$me_3(x)$	$me_4(x)$	Φ	$\dot{\mathbf{if}}\ me_1(x)\cdots me_4(x)\ \Phi$
$-$	$+$	0	0	$e_3(\bot) \sqsupseteq e_4(\top)$	$+$
$+$	$-$	0	0	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
$+$	0	0	0	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
0	0	0	0	irrelevant	0

Note, for example, that the (IF) rule can be instantiated to:

$$\frac{x:\tau' \vdash e_1:\tau,\{x\mapsto -\} \quad x:\tau' \vdash e_2:\tau,\{x\mapsto +\}}{x:\tau' \vdash \mathit{if}\ e_1 \sqsubseteq e_2\ \mathit{then}\ \top\ \mathit{else}\ \bot : \iota,\{x\mapsto +\}}$$

$$\frac{x:\tau' \vdash e_1:\tau,\{x\mapsto +\} \quad x:\tau' \vdash e_2:\tau,\{x\mapsto -\}}{x:\tau' \vdash \mathit{if}\ e_1 \sqsubseteq e_2\ \mathit{then}\ \top\ \mathit{else}\ \bot : \iota,\{x\mapsto -\}}$$

and further to:

$$\frac{x:\tau \vdash x:\tau,\{x\mapsto +\} \quad x:\tau \vdash \bot:\tau,\bar{0}}{x:\tau \vdash \mathit{if}\ x \sqsubseteq \bot\ \mathit{then}\ \top\ \mathit{else}\ \bot : \iota,\{x\mapsto -\}}$$

We can sharpen the (IF) rule for the case in which the condition is of the special shape $x \sqsubseteq c$, which actually occurs quite frequently in program analysis specifications. Here the true-false boundary is clearly known and we exploit that in order to define **ifc**:

$$\frac{\Gamma \vdash e_3:\tau, me_3 \quad \Gamma \vdash e_4:\tau, me_4}{\Gamma \vdash \mathit{if}\ x \sqsubseteq c\ \mathit{then}\ e_3\ \mathit{else}\ e_4 : \tau, \mathbf{ifc}\ x\ me_3\ me_4\ \Phi} \qquad \text{(IFC)}$$

When we increase x, the value switches from e_3 to e_4 at points directly above c in the associated lattice. Let $\widehat{c}$ be the set of such elements:

$$\widehat{c} = \{\ x \mid x \sqsupset c,\ \forall y.(x \sqsupseteq y \sqsupset c \Rightarrow x = y)\}\ .$$

We can gain more precision if we use $\widehat{c}$ to determine whether switches preserve monotonicity. For instance, suppose e_3 and e_4 have one free variable and both are monotonic with respect to it. Then if $e_3(c)$ does not exceed $e_4(d)$ for every $d \in \widehat{c}$, then the whole *if*-expression is also monotonic. Hence, we can define **ifc** to be

$$\mathbf{ifc}\ x\ me_3\ me_4\ \Phi = \{y \mapsto \begin{cases} \dot{\mathbf{ifc}}\ me_3(y)\ me_4(y)\ \Phi, & \text{if } y = x \\ me_3(y) \sqcup me_4(y), & \text{otherwise} \end{cases} \mid y \in \mathit{Var}\}$$

where the definition of $\dot{\mathbf{ifc}}$ is (assuming that e_3 and e_4 have only one free variable) :

$me_3(x)$	$me_4(x)$	Φ	$\dot{\mathbf{ifc}}\ me_3(x)\ me_4(x)\ \Phi$
$+$	$+$	$\forall d \in \widehat{c}.\ e_3(c) \sqsubseteq e_4(d)$	$+$
$+$	0	$\forall d \in \widehat{c}.\ e_3(c) \sqsubseteq e_4(d)$	$+$
0	$+$	$\forall d \in \widehat{c}.\ e_3(c) \sqsubseteq e_4(d)$	$+$
$-$	$-$	$\forall d \in \widehat{c}.\ e_3(c) \sqsupseteq e_4(d)$	$-$
$-$	0	$\forall d \in \widehat{c}.\ e_3(c) \sqsupseteq e_4(d)$	$-$
0	$-$	$\forall d \in \widehat{c}.\ e_3(c) \sqsupseteq e_4(d)$	$-$
0	0	irrelevant	0

If e_3,e_4 have occurrences of more variables, one should use c and d with a combination of $\bot$ and $\top$ depending on the monotonicity of e_3 and e_4 with respect to the other variables. The Φ condition is computable only when the set $\widehat{c}$ of the associated lattice is finite.

4 Soundness

First we introduce some notation. For $s, s' \in ME$ we write $s \sqsubseteq s'|_x$ iff $s(x) \sqsubseteq s'(x)$ and $s(y) = s'(y)$ for $y \neq x$. Given lattice elements v, v', a monotonicity behavior $me \in ME$, and a variable x we define $v\ me(x)\ v'$ by:

$$v\ me(x)\ v = \begin{cases} v = v', & me(x) = 0 \\ v \sqsupseteq v', & me(x) = + \\ v \sqsubseteq v', & me(x) = - \end{cases}$$

Next let $v : \tau$ be a logical relation between lattice elements and types satisfying:

$$\begin{array}{ll} c : \iota & \text{iff true} \\ (\lambda x.e, s) : \tau_1 \stackrel{me}{\to} \tau_2 & \text{iff} \\ & (1)\ v_1 : \tau_1 \text{ and } s + x : v_1 \vdash e \Rightarrow v_2 \text{ implies } v_2 : \tau_2 \\ & (2)\ v_1 : \tau_1,\ v_1' : \tau_1,\ v_1 \sqsubseteq v_1',\ s + x : v_1 \vdash e \Rightarrow v_2, \\ & \quad s + x : v_1' \vdash e \Rightarrow v_2' \text{ implies } v_2\ me(x)\ v_2' \end{array}$$

where $s \vdash e \Rightarrow v$ means that v is the result of evaluating e in the value environment s. We write $s \models \Gamma$ when the value environment s respects the type environment Γ:

$$\frac{}{\emptyset \models \emptyset} \qquad \frac{s \models \Gamma \quad v : \tau}{s + x : v \models \Gamma + x : \tau}$$

Finally we are ready to state the correctness result:

Theorem 1 *If* $\Gamma \vdash e : \tau, me$ *then*

(A) $s \models \Gamma$ *and* $s \vdash e \Rightarrow v$ *imply* $v : \tau$

(B) $s \models \Gamma,\ s' \models \Gamma,\ s \sqsubseteq s'|_x,\ s \vdash e \Rightarrow v,$ *and* $s' \vdash e \Rightarrow v'$ *imply* $v\ me(x)\ v'$

Proof. By structural induction on e.
Case $\Gamma \vdash \lambda x.e : \tau_1 \stackrel{me}{\to} \tau_2, me[0/x]$.

(A) Let $s \models \Gamma$ and $s \vdash \lambda x.e \Rightarrow (\lambda x.e, s)$.
We have to show $(\lambda x.e, s) : \tau_1 \stackrel{me}{\to} \tau_2$.
By definition $\Gamma + x : \tau_1 \vdash e : \tau_2, me'$ and $me' \sqsubseteq me$.
Let $v_1 : \tau_1$. Then $s + x : v_1 \models \Gamma + x : \tau_1$.
Suppose $s + x : v_1 \vdash e \Rightarrow v_2$. Then, by IH, $v_2 : \tau_2$, hence condition (1) holds.

For (2) suppose $v_1' : \tau_1$ and $v_1 \sqsubseteq v_1'$ and $s{+}x : v_1 \vdash e \Rightarrow v_2$ and $s{+}x : v_1' \vdash e \Rightarrow v_2'$.
By IH, $v_2\ me'(x)\ v_2'$, i.e. $v_2\ me(x)\ v_2'$ because $me' \sqsubseteq me$.
Thus the condition (2) holds.

(B) Let $s \models \Gamma$, $s' \models \Gamma$, $s \sqsubseteq s'|_y$,
$s \vdash \lambda x.e \Rightarrow (\lambda x.e, s)$, and $s' \vdash \lambda x.e \Rightarrow (\lambda x.e, s')$.
We have to show: $(\lambda x.e, s)\ me[0/x](y)\ (\lambda x.e, s')$.
i.e. to show: $s + x : v_1 \vdash e \Rightarrow v_2$ and $s' + x : v_1 \vdash e \Rightarrow v_2'$
implies $v_2\ me[0/x](y)\ v_2'$

- When $y = x$.
  Then $s + x : v_1 = s' + x : v_1$.
  Thus $s + x : v_1 \vdash e \Rightarrow v_2$ and $s' + x : v_1 \vdash e \Rightarrow v_2'$
  imply $v_2 = v_2'$, thus $v_2\ (me[0/x](y))\ v_2'$.
- When $y \neq x$.
  By definition, $\Gamma + x : \tau_1 \vdash e, b : me', \text{and } me' \sqsubseteq me$.
  Observe that $s + x : v_1 \sqsubseteq s' + x : v_1|_y$,
  $s + x : v1 \models \Gamma + x : \tau_1$, and $s' + x : v1 \models \Gamma + x : \tau_1$.
  Let $s + x : v_1 \vdash e \Rightarrow v_2$ and $s' + x : v_1 \vdash e \Rightarrow v_2'$.
  Then by IH, $v_2\ me'(y)\ v_2'$, i.e. $v_2\ me(y)\ v_2'$ because $me' \sqsubseteq me$,
  so $v_2\ (me[0/x](y))\ v_2'$.

The reasoning in other cases is pretty much similar and uses the arguments we have outlined informally when introducing the system. □

5 Algorithm

Our effect-type system is a little different from conventional effect systems. In [TT94, TT93, TJ92, TJ91] effects are constant symbols and the only operation involved is set-union. In this paper effects (monotonicity tables) are subject to other operations: $\otimes$, **if** and **ifc**. Hence, we cannot solely rely on the unification procedure [Rob65] for type inference.

Our algorithm consists of two phases: we derive constraints for types and monotonicity effects first, then we solve the constraints. There are two kinds of constraints: for types (τ) and for monotonicity behaviors (me). The type constraints will be solved by unification [Rob65] and the monotonicity constraints – by fixpoint iteration. Unification is applicable to the type constraints because they are simply equality constraints with variables for the latent-effects. Its result will provide us with some additional monotonicity constraints about the latent effects of function types. Then conventional fixpoint iteration can be applied to the monotonicity constraints since every operator (@, **if**, and **ifc**) on the constraints is monotonic. Because the least model for the constraints is equivalent to the least fixed point of the corresponding equations [CC95], the algorithm will give the best approximation of monotonicity that could be inferred in our type system.

5.1 Extraction of Constraints

Each constraint ρ will be a monotonicity formula constructed according to the following rules:

$$\begin{array}{rcl}\rho & ::= & \tau_1 \doteq \tau_2 \mid me_1 \supseteq me_2 \\ & | & \exists\alpha.\rho \mid \exists\beta.\rho \\ & | & \rho_1, \rho_2\end{array}$$

where variables are allowed to occur in types τ and monotonicity behaviors me:

$$\begin{array}{lcll}\tau & ::= & \text{as before} \mid \alpha & \text{(type variable)} \\ me & ::= & \text{as before} \mid \beta & \text{(monotonicity variable)}\end{array}$$

We write α_i for type variables, and β_i for monotonicity variables. The validity $\vdash \rho$ of the formula ρ is defined as follows. $\{x/y\}\rho$ denotes ρ in which x has been substituted for y.

$$\frac{}{\vdash \tau \doteq \tau} \qquad \frac{me_1 \sqsupseteq me_2}{\vdash me_1 \supseteq me_2} \qquad \frac{\vdash \{\tau/\alpha\}\rho}{\vdash \exists\alpha.\rho} \qquad \frac{\vdash \{me/\beta\}\rho}{\vdash \exists\beta.\rho} \qquad \frac{\vdash \rho_1 \quad \vdash \rho_2}{\vdash \rho_1, \rho_2}$$

We extract the associated monotonicity formula from an expression e using a recursive procedure $C(\Gamma, e, \tau, me)$. It has linear time complexity (with respect to the size of e). The size of the generated formula is also linear in e's size:

$$\begin{array}{rcl}
C(\Gamma, c, \tau, me) & = & \tau \doteq \iota, \quad me \supseteq \bar{0} \\
C(\Gamma, x, \tau, me) & = & \tau \doteq \Gamma(x), \quad me \supseteq \{x \mapsto +\} \\
C(\Gamma, \lambda x.e, \tau, me) & = & \exists \alpha_1 \alpha_2 \beta_1 \beta_2. \\
& & C(\Gamma + x : \alpha_1, e, \alpha_2, \beta_1), \\
& & \tau \doteq \alpha_1 \stackrel{\beta_2}{\rightarrow} \alpha_2, \quad \beta_2 \supseteq \beta_1, \\
& & me \supseteq \beta_2[0/x] \\
C(\Gamma, \text{fix } f\ e, \tau, me) & = & \exists \beta. \\
& & C(\Gamma + f : \tau, e, \tau, \beta) \\
& & me \supseteq \beta[0/f] \\
C(\Gamma, e_1\ e_2, \tau, me) & = & \exists \alpha \beta_1 \beta_2 \beta_3. \\
& & C(\Gamma, e_1, \alpha \stackrel{\beta_1}{\rightarrow} \tau, \beta_2), \quad C(\Gamma, e_2, \alpha, \beta_3), \\
& & me \supseteq @\ \beta_1\ \beta_2\ \beta_3 \\
C(\Gamma, e_1 \sqcup e_2 \text{ or } e_1 \sqcap e_2, \tau, me) & = & \exists \beta_1 \beta_2. \\
& & C(\Gamma, e_1, \tau, \beta_1), \quad C(\Gamma, e_2, \tau, \beta_2), \\
& & me \supseteq \beta_1 \sqcup \beta_2 \\
C(\Gamma, \mathit{if}\ e_1 \sqsubseteq e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4, \tau, me) & = & \exists \alpha \beta_1 \beta_2 \beta_3 \beta_4. \\
& & C(\Gamma, e_1, \alpha, \beta_1), \quad C(\Gamma, e_2, \alpha, \beta_2), \\
& & C(\Gamma, e_3, \tau, \beta_3), \quad C(\Gamma, e_4, \tau, \beta_4), \\
& & me \supseteq \mathbf{if}\ \beta_1\ \beta_2\ \beta_3\ \beta_4\ \Phi \\
C(\Gamma, \mathit{if}\ x \leq c\ \mathit{then}\ e_3\ \mathit{else}\ e_4, \tau, me) & = & \exists \beta_3 \beta_4. \\
& & C(\Gamma, e_3, \tau, \beta_3), \quad C(\Gamma, e_4, \tau, \beta_4), \\
& & me \supseteq \mathbf{ifc}\ \beta_3\ \beta_4\ \Phi
\end{array}$$

It is easy to see that the validility of the generated formula $C(\Gamma, e, \tau, me)$ is equivalent to the typing judgment $\Gamma \vdash e : \tau, me$. Below we give part of the proof in the case of lambda expressions.

Theorem 2 $\vdash C(\Gamma, e, \tau, me)$ *iff* $\Gamma \vdash e : \tau, me$.

Proof. By structural induction on e.
Case $\lambda x.e$.

$\Rightarrow$ There are τ_1, τ_2, b, b' such that
$C(\Gamma + x : \tau_1, e, \tau_2, b')$, $b' \sqsubseteq b$, $me = b[0/x]$ and $\tau = \tau_1 \xrightarrow{b} \tau_2$.
By IH, $\Gamma + x : \tau_1 \vdash e : \tau_2, b'$.
By (LAM),$\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{b} \tau_2, b[0/x]$,
i.e. $\Gamma \vdash \lambda x.e : \tau, me$.

$\Leftarrow$ By (LAM), $\Gamma \vdash \lambda x.e : \tau, me$.
It implies $\Gamma + x : \tau_1 \vdash e : \tau_2, b'$, $b' \sqsubseteq b$, $\tau = \tau_1 \xrightarrow{b} \tau_2$ and $me = b[0/x]$.
By IH, $\vdash C(\Gamma + x : \tau_1, e, \tau_2, b')$,
i.e. $\exists \alpha_1 \alpha_2 \beta \beta'.C(\Gamma + x : \alpha_1, e, \alpha_2, \beta')$ and $\beta' \sqsubseteq \beta$ and $\alpha_1 \xrightarrow{\beta} \alpha_2$ and $me = \beta[0/x]$.

□

5.2 Solving the Constraints

We observe two properties of the generated monotonicity formula $C(\Gamma, e, \alpha, \beta)$. Firstly, in every occurrence of $\tau_1 \xrightarrow{me} \tau_2$, me will be one of the variables β_i. This is quite obvious, because the property holds at the only two places where such a latent type is formed (lambda abstraction and application). Secondly, every me_1 in $me_1 \supseteq me_2$ is a variable β_i. This is because for every generated monotonicity constraint $me_1 \supseteq me_2$ the left-hand-side me_1 is the last parameter to C, which is a monotonicity variable β_i for every recursive call to C.

Thanks to the first property, the unification procedure can be applied to the type constraints $\{\tau \doteq \tau' \in C(\Gamma, e, \alpha, \beta)\}$, and the resultant substitution involves only monotonicity variables β_i.

Each item β'/β in the substitution is equivalent to the monotonicity contraints $\beta' \supseteq \beta$ and $\beta \supseteq \beta'$. This set of unification-driven contraints, together with the monotonicity contraints from $C(\Gamma, e, \alpha, \beta)$, constitute the equations (e.g. "$\beta \supseteq me_1, \cdots, \beta \supseteq me_k$" as "$\beta = me_1 \sqcup \cdots \sqcup me_k$") whose least solution corresponds to the the least model of the original contraints [CC95]. The least solution is computed by iteration: starting from $\bar{0}$ for every me_i we repeatedly apply the right-hand-sides of the equations to the intermediate result. This procedure terminates with the least fixed point, because the operators involved ($@$, **if**, **ifc**, $\sqcup$) are all monotonic.
Complexity. The constraint extraction procedure C takes linear time in the size of the input program. The number of generated constraints (type equations and monotonicity constraints) is also linear. Then the unification takes linear time with respect to the number of type equations. Because there are $O(n)$ indeterminates (me_i) where n is the program size, the iteration will take $O(n^2)$ steps in the worst case, since no chain in the lattice $Var \to \{0, +, -, \top\}$ can be longer than $2 \times |Var|$ (Var is here the finite set of free variables occurring in the input program). The overall time complexity is therefore $O(n^3)$, because each equation computes a new monotonicity behavior me_i whose size is $O(|Var|)$ and constant time is needed for table look-up for each operator.

6 Conclusion

Our work provides a method of monotonicity verification for λ-definable functions over arbitrary finite-height lattices. Static monotonicity analysis seems an interesting problem on its own and apparently not much work has been done in that area. Our interest in this topic was motivated by Zoo [Yi01a, Yi01b], which is a program-analyzer generator. Now that it can automatically check whether the input specification is monotonic or not, termination of the specified analysis is guaranteed if the outcome of the test is positive. Thus we can prevent Zoo from generating divergent analyzers.

Our verification procedure is an effect-type system, which can be classified as monovariant flow-insensitive analysis. Its effectiveness remains to be investigated and experiments are underway for existing program analyses (e.g. conventional data flow analyses [ASU86, YH93] and exception analyses [YR01, YR97]). We would also like to make the rules more liberal by employing other static analysis tools to estimate the boundary region in the conditional expression. It seems that there is not too much scope for improvement in the rest of cases.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[CC77] Patrick Cousot and Radhia 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*, pages 238–252, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992. Also as a tech report: Ecole Polytechnique, no. LIX/RR/92/10.

[CC95] Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Lecture Notes in Computer Science*, volume 939, pages 293–308. Springer-Verlag, proceedings of the 7th international conference on computer-aided verification edition, 1995.

[DGL+99] Yevgeniy Dodis, Oded Goldreich, Eric Lehman, Sofya Raskhodnikova, Dana Ron, and Alex Samorodnitsky. Improved testing algorithms for monotonicity. Number Report TR99-107 in Electronic Colloquium on Computational Compleity, June 1999.

[GGLR98] Oded Goldreich, Shafi Goldwasser, Eric Lehman, and Dand Ron. Testing monotonicity. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, 1998.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rut65] D.E. Rutherford. *Introduction to Lattice Theory.* Hafner Publishing Company, New York, 1965.

[Sch96] Winfrid G. Schneeweiss. A necessary and sufficient criterion for the monotonicity of boolean functions with deterministic and stochastic. *IEEE Transactions on Computers*, 45(11):1300–1302, November 1996.

[TJ91] Jean-Pierre Talpin and Pierre Jouvelot. Type, effect and region reconstruction in polymorphic functional languages. In *Proceedings of Functional Programming Languages and Computer Architecture*, 1991.

[TJ92] Jean-Piere Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[TT93] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report Technical Report 93/15, Department of Computer Science, Copenhagen University, July 1993.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

[Vor00] Andrei Voronenko. On the complexity of the monotonicity verification. In *Proceedings of the 15th Annual IEEE Conference on Computational Complexity*, pages 4–7, July 2000.

[YH93] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, January 1993.

[Yi01a] Kwangkeun Yi. *Program Analysis System Zoo.* Research On Program Analysis: National Creative Research Center, KAIST, July 2001. `http://ropas.kaist.ac.kr/zoo/doc/rabbit-e.ps`.

[Yi01b] Kwangkeun Yi. System Zoo: towards a realistic program analyzer generator, July 2001. Seminar talk at ENS, Paris. `http://ropas.kaist.ac.kr/~kwang/talk/ens01/ens01.ps`.

[YR97] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 1997.

[YR01] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 273(1), 2001. (to appear).

A Operator Definition

Full definition of $\dot{\textbf{if}}$, assuming only a single variable can occur freely in e_3 and e_4. Note that the operation is monotonic. For missing cases, the results are equal to $\top$.

$me_1(x)$	$me_2(x)$	$me_3(x)$	$me_4(x)$	Φ	$\dot{\textbf{if}}\ me_1(x)\cdots me_4(x)\ \Phi$
−	+	+	+	$e_3(\bot) \sqsupseteq e_4(\top)$	+
0	+	+	+	$e_3(\bot) \sqsupseteq e_4(\top)$	+
−	0	+	+	$e_3(\bot) \sqsupseteq e_4(\top)$	+
−	+	0	+	$e_3(\bot) \sqsupseteq e_4(\top)$	+
−	+	+	0	$e_3(\bot) \sqsupseteq e_4(\top)$	+
0	0	+	+	irrelevant	+
0	+	0	+	$e_3(\bot) \sqsupseteq e_4(\top)$	+
0	+	+	0	$e_3(\bot) \sqsupseteq e_4(\top)$	+
−	0	0	+	$e_3(\bot) \sqsupseteq e_4(\top)$	+
−	0	+	0	$e_3(\bot) \sqsupseteq e_4(\top)$	+
−	+	0	0	$e_3(\bot) \sqsupseteq e_4(\top)$	+
−	0	0	0	$e_3(\bot) \sqsupseteq e_4(\top)$	+
0	+	0	0	$e_3(\bot) \sqsupseteq e_4(\top)$	+
0	0	+	0	irrelevant	+
0	0	0	+	irrelevant	+
+	−	+	+	$e_3(\top) \sqsubseteq e_4(\bot)$	+
0	−	+	+	$e_3(\top) \sqsubseteq e_4(\bot)$	+
+	0	+	+	$e_3(\top) \sqsubseteq e_4(\bot)$	+
+	−	0	+	$e_3(\top) \sqsubseteq e_4(\bot)$	+
+	−	+	0	$e_3(\top) \sqsubseteq e_4(\bot)$	+
0	−	0	+	$e_3(\top) \sqsubseteq e_4(\bot)$	+
0	−	+	0	$e_3(\top) \sqsubseteq e_4(\bot)$	+
+	0	0	+	$e_3(\top) \sqsubseteq e_4(\bot)$	+
+	0	+	0	$e_3(\top) \sqsubseteq e_4(\bot)$	+
+	−	0	0	$e_3(\top) \sqsubseteq e_4(\bot)$	+
+	0	0	0	$e_3(\top) \sqsubseteq e_4(\bot)$	+
0	−	0	0	$e_3(\top) \sqsubseteq e_4(\bot)$	+
−	+	−	−	$e_3(\bot) \sqsubseteq e_4(\top)$	−
0	+	−	−	$e_3(\bot) \sqsubseteq e_4(\top)$	−
−	0	−	−	$e_3(\bot) \sqsubseteq e_4(\top)$	−
−	+	0	−	$e_3(\bot) \sqsubseteq e_4(\top)$	−
−	+	−	0	$e_3(\bot) \sqsubseteq e_4(\top)$	−
0	0	−	−	irrelevant	−
0	+	0	−	$e_3(\bot) \sqsubseteq e_4(\top)$	−
0	+	−	0	$e_3(\bot) \sqsubseteq e_4(\top)$	−
−	0	0	-	$e_3(\bot) \sqsubseteq e_4(\top)$	−
−	0	-	0	$e_3(\bot) \sqsubseteq e_4(\top)$	−
-	+	0	0	$e_3(\bot) \sqsubseteq e_4(\top)$	−
−	0	0	0	$e_3(\bot) \sqsubseteq e_4(\top)$	−
0	+	0	0	$e_3(\bot) \sqsubseteq e_4(\top)$	−
0	0	−	0	irrelevant	−
0	0	0	−	irrelevant	−

(continued)

$me_1(x)$	$me_2(x)$	$me_3(x)$	$me_4(x)$	Φ	$\dot{\textbf{if}}\ me_1(x) \cdots me_4(x)\ \Phi$
$+$	$-$	$-$	$-$	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
0	$-$	$-$	$-$	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
$+$	0	$-$	$-$	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
$+$	$-$	0	$-$	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
$+$	$-$	$-$	0	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
0	$-$	0	$-$	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
0	$-$	$-$	0	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
$+$	0	0	$-$	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
$+$	0	$-$	0	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
$+$	-	0	0	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
$-$	0	0	0	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
0	$-$	0	0	$e_3(\top) \sqsupseteq e_4(\bot)$	$-$
0	0	0	0	irrelevant	0