
ROPAS

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

ROPAS MEMO
2002-17
September 18, 2002

Static Extensionality Analysis for λ-Definable Functions Over Lattices *†

Kwangkeun Yi and Hyunjun Eo
{kwang; poisson}@ropas.kaist.ac.kr
ROPAS‡
Dept. of Computer Science
Korea Advanced Institute of Science & Technology

September 18, 2002

Abstract

We employ static analysis to examine extensionality ($\forall x : x \leq f(x)$) 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 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 extensionality of the functions is ascertained, the generated analyzer is guaranteed to terminate. In a disjunctive combination with the previous work [MY02] on the static monotonicity analysis (checking $\forall x \leq y : f(x) \leq f(y)$), the extensionality analysis will enlarge the set of input programs accepted by Zoo.

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, CC79]. 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 (encoded in C and ML) which, given an input program to analyze, derives a set of data-flow equations and solves them by fixpoint iterations.

One of our goals is to enable Zoo to check whether the user-specified abstract interpreter defines a correct and terminating analysis. We don't want Zoo to blindly generate an executable program without verifying that the input specification qualifies

*This work is done while the first author was visiting the Computer Science Department, École Normale Supérieure, 45 rue d'Ulm, Paris, France and is partially 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.

‡Research On Program Analysis System (ropas.kaist.ac.kr), National Creative Research Initiative Center, KAIST, Korea.

for static analysis. We want the job of assuring the correctness of the specified abstract interpreter to be, if possible, automatically carried out by Zoo.

Towards this goal, we have designed a static analysis method by which Zoo can check the extensionality of the input abstract interpreters (Rabbit programs). The extensionality is a sufficient condition for the generated analyzers always terminate. An abstract interpreter consists of lattice definitions and definitions of functions over the lattices. Once it is known that the functions are extensional ($\forall x : x \leq f(x)$), the generated analyzers are guaranteed to terminate (assuming that the analyzer ranges over finite-height lattices). The generated analyzer computes the sequence $\bot$, $f(\bot)$, $f(f(\bot)), \cdots$ and the extensionality of f guarantees that the sequence reaches a stable point. By using this analysis, Zoo can statically estimate the extensionality of the input functions and consequently reject analyzers whose specification is possibly not extensional.

In a disjunctive combination with the previous work [MY02] on the static monotonicity analysis(checking $\forall x \leq y : f(x) \leq f(y)$), the extensionality analysis will enlarge the set of Rabbit programs accepted by Zoo. This is because the extensionality and the monotonicity are incomparable properties and either of them guarantees the termination of the generated analyzers that are defined over finite-height lattices. For example, "$\lambda x.\mathit{if}\ x \sqsubseteq c\ \mathit{then}\ \bot\ \mathit{else}\ c$" is monotonic but not extensional, while "$\lambda x.\mathit{if}\ x = c\ \mathit{then}\ c\ \mathit{else}\ \top$" is extensional but not monotonic. Yet, each of the fixpoint-iteration chain of the two functions reach to a fixpoint in a finite time: the former to $\bot$ in one iteration and the latter to $\top$ in two iterations. The readers who wonder how just extensional static analysis is proven correct may want to see appendix A.

2 Setting

Let L be a lattice. A function $f : L \longrightarrow L$ is extensional (respectively anti-extensional) if and only if for all x, we have $x \leq f(x)$ (respectively $f(x) \leq x$). If a function is both extensional and anti-extensional, it is the identical function. For functions from product lattices (multiple arguments), we can analogously define extensionality and anti-extensionality with respect to the ith component (ith argument).

Our goal is to design a static procedure that can certify whether a function between two lattices is extensional or not. The source language is Rabbit [Yi01a], the input specification language of the Zoo system. Rabbit is a strongly-typed, statically-scoped, functional eager-evaluation language. Rabbit's core is defined as:

e	::=	c	constant (lattice point)
	|	x	variable
	|	$\lambda x.e$	function
	|	$Y\,\lambda x.e$	recursive function
	|	$e\ e$	application
	|	$e \sqcup e$	join operation
	|	$e \sqcap e$	meet operation
	|	$\mathit{if}\ e \sqsubseteq e\ ?\ e\ :\ e$	branching

Values in this language are either lattice elements or functions over lattices. Hence a type is either a lattice L or a function $\tau \rightarrow \tau$ between types. c is a constant

$$\text{Value} \qquad v \;\rightarrow\; c \mid x \mid \lambda x.e \mid Y\,v$$

$$\frac{}{c \hookrightarrow c} \qquad \frac{}{x \hookrightarrow x}$$

$$\frac{}{\lambda x.e \hookrightarrow \lambda x.e} \qquad \frac{}{Y\lambda x.e \hookrightarrow Y\lambda x.e}$$

$$\frac{e_1 \hookrightarrow \lambda x.e \quad e_2 \hookrightarrow v' \quad \{v'/x\}e \hookrightarrow v''}{e_1\ e_2 \hookrightarrow v''} \qquad \frac{e_1 \hookrightarrow Y\,v \quad e_2 \hookrightarrow v' \quad v(Y\,v) \hookrightarrow \lambda x.e \quad \{v'/x\}e \hookrightarrow v''}{e_1\ e_2 \hookrightarrow v''}$$

$$\frac{\forall i \in \{1,2\} : e_i \hookrightarrow c_i}{e_1 \sqcup e_2 \hookrightarrow c_1 \sqcup c_2} \qquad \frac{\forall i \in \{1,2\} : e_i \hookrightarrow c_i}{e_1 \sqcap e_2 \hookrightarrow c_1 \sqcap c_2}$$

$$\frac{\forall i \in \{1,2\} : e_i \hookrightarrow v_i \quad v_1 \sqsubseteq v_2 \quad e_3 \hookrightarrow v_3}{\mathit{if}\ e_1 \sqsubseteq e_2\ ?\ e_3\ :\ e_4 \hookrightarrow v_3} \qquad \frac{\forall i \in \{1,2\} : e_i \hookrightarrow v_i \quad v_2 \not\sqsubseteq v_2 \quad e_4 \hookrightarrow v_4}{\mathit{if}\ e_1 \sqsubseteq e_2\ ?\ e_3\ :\ e_4 \hookrightarrow v_4}$$

Figure 1: Evaluation rules

expression denoting a lattice element. The *if* expression branches, as usual, depending on whether the conditional partial-order relation holds or not. The usual evaluation rule for $e \hookrightarrow v$(expression e computes v) is in Figure 1. Notation $\{v/x\}e$ denotes the result from substituting v for free variable x in e.

In the actual Rabbit language [Yi01a], user-definable lattices are product lattices, powerset lattices, function lattices, and lattices with user-defined orders.

Throughout this paper we assume that every variable is uniquely named and a unique mono-type is associated to each expression. We also assume that there is no dead code in a program, i.e., there is no function which can not be applied.

3 Extensionality Checking by a Deductive System

Given an expression e of the core language, our extensionality check will determine conservatively whether the operation

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

is extensional, anti-extensional, or constant with respect to each of its free variables $x_1, \cdots, x_n$.

Example 1 *"$(x \sqcap c) \sqcup y$" is at least as large as y hence extensional for y but unknown for x. "if $x \sqsubseteq c\ ?\ \top\ :\ x \sqcup y$" is at least as large as both x and y hence extensional for both x and y.*

We present the checking procedure as an inference system for judgments of the form

$$\Gamma \vdash e : \mathsf{xb}.$$

The judgments should be read as "under assumption Γ, expression e has extensionality behavior xb".

Extensionality assumption Γ and behavior xb range over the following domains:

Definition 1

$$\begin{array}{rcll}
\Gamma & \in & \mathit{XE} & = & \mathit{Var} \xrightarrow{\mathit{fin}} \mathit{XB} & \textit{extensionality assumption} \\
\mathsf{xb} & \in & \mathit{XB} & = & X + \mathit{XB} \times \mathit{XB} & \textit{extensionality behavior} \\
\mathsf{x} & \in & X & = & \sum_{\tau \in \mathit{Type}} (\mathit{Var}_\tau \xrightarrow{\mathit{fin}} T) & \textit{non-functional extensionality} \\
\mathsf{xb} \to \mathsf{xb} & \in & & & \mathit{XB} \times \mathit{XB} & \textit{functional extensionality} \\
t & \in & T & = & \{\bot, 0, +, -, \top\} & \textit{extensionality token} \\
x, y, z & \in & \mathit{Var} & & & \textit{variables} \\
 & & \mathit{Var}_\tau & & & \textit{variables of type } \tau \\
\tau & \in & \mathit{Type} & & & \textit{program types}
\end{array}$$

The extensionality assumption

$$\Gamma \in \mathit{Var} \xrightarrow{\text{fin}} \mathit{XB}$$

is a finite map from program variables to extensionality behaviors XB. The extensionality behavior xb is either a pair

$$\mathsf{xb} \to \mathsf{xb} \in \mathit{XB} \times \mathit{XB}$$

of extensionality behaviors for function-typed expressions, or a map

$$\mathsf{x} \in X = \sum_{\tau \in \mathit{Type}} (\mathit{Var}_\tau \xrightarrow{\text{fin}} T)$$

from variables of type τ to extensionality tokens. This typefulness of XB (constructively implied by the typefulness of X) is eligible because every expression is uniquely typed and its extensionality is, by definition, in terms of variables of the same type.

For $\mathsf{xb} \in \mathit{XB}$ or $a \in \mathit{Var}$, we sometimes write $\mathsf{xb}_{\mathit{type}(\mathsf{xb})}$ or $x_{\mathit{type}(x)}$ to expose their types.

Definition 2 *The extensionality behavior* xb *has the following meaning* $[\![\mathsf{xb}]\!]$*:*

$$\begin{array}{rcl}
[\![\mathsf{x}]\!] & = & \cap\{[\![x^t]\!] \mid \mathsf{x}(x) = t\} \\
[\![\mathsf{xb}_1 \to \mathsf{xb}_2]\!] & = & \{e \mid \forall e_1 \in [\![\mathsf{xb}_1]\!] : e\, e_1 \in [\![\mathsf{xb}_2]\!]\} \\
\textit{where} & & \\
[\![x^0]\!] & = & \{e \mid x \in \mathit{FV}(e) \wedge (\forall s : \{s\}e \hookrightarrow v \textit{ implies } s(x) = v)\} \\
[\![x^+]\!] & = & \{e \mid x \in \mathit{FV}(e) \wedge (\forall s : \{s\}e \hookrightarrow v \textit{ implies } s(x) \sqsubseteq v)\} \\
[\![x^-]\!] & = & \{e \mid x \in \mathit{FV}(e) \wedge (\forall s : \{s\}e \hookrightarrow v \textit{ implies } v \sqsubseteq s(x))\} \\
[\![x^\top]\!] & = & \textit{all expressions} \\
[\![x^\bot]\!] & = & \emptyset
\end{array}$$

Note the extensionality tokens X form a lifted diamond-shaped lattice:

$$\bot \sqsubseteq 0 \sqsubseteq + \sqsubseteq \top \quad \text{and} \quad \bot \sqsubseteq 0 \sqsubseteq - \sqsubseteq \top.$$

The partial ordern in XB follows from the meanings of XB elements:

Definition 3 *The order in X is point-wise:*

$$\mathsf{x}_\tau \sqsubseteq \mathsf{x}'_\tau \;\; \textit{iff} \;\; \forall x \in \mathit{Var}_\tau : \mathsf{x}(x) \sqsubseteq \mathsf{x}'(x)$$

and the order in $XB \times XB$ is contra-variant on first component:

$$\mathsf{xb}_1 \to \mathsf{xb}_2 \sqsubseteq \mathsf{xb}'_1 \to \mathsf{xb}'_2 \;\; \textit{iff} \;\; \mathsf{xb}'_1 \sqsubseteq \mathsf{xb}_1 \;\wedge\; \mathsf{xb}_2 \sqsubseteq \mathsf{xb}'_2.$$

For constant expression, we have three cases, $\bot$, $\top$, and other constant c:

$$\frac{}{\Gamma \vdash \bot_\tau : \{x \mapsto - \mid x \in \mathit{Var}_\tau\}}\ (\text{CON-1}) \qquad \frac{}{\Gamma \vdash \top_\tau : \{x \mapsto + \mid x \in \mathit{Var}_\tau\}}\ (\text{CON-2})$$

$$\frac{}{\Gamma \vdash c_\tau : \{x \mapsto \top \mid x \in \mathit{Var}_\tau\}}\ (\text{CON-3})$$

A variable's extensionality is assumed in the environment:

$$\frac{\Gamma(x) = \mathsf{xb}'}{\Gamma \vdash x : \mathsf{xb}}\ (\text{VAR})$$

The extensionality of the join operation is compositional:

$$\frac{\Gamma \vdash e_1 : \mathsf{xb}_1 \quad \Gamma \vdash e_2 : \mathsf{xb}_2}{\Gamma \vdash e_1 \sqcup e_2 : \mathsf{xb}_1 \oplus \mathsf{xb}_2}\ (\text{LUB})$$

Note that the extensionality of the two subexpressions is reflected by the extensionality of the whole term via the $\oplus$ operation. Let e_i be of type τ, that is, the xb_i are maps from τ-typed variables. The $\oplus$ operation over X is point-wise:

$$\mathsf{x}_\tau \oplus \mathsf{x}'_\tau \;\; = \;\; \{x \mapsto (\mathsf{x}(x) \oplus \mathsf{x}'(x)) \mid x \in \mathit{Var}_\tau\}$$

The commutative $\oplus$ for the extensionality tokens takes the effect of moving-up($\sqcup$) operation into account:

$\oplus$	0	$+$	$-$	$\top$
0	0	$+$	0	$+$
$+$		$+$	$+$	$+$
$-$			$-$	$\top$
$\top$				$\top$

The correctness of $\oplus$ for tokens is easy to see. For example, if e_i is extensional for x ($x \sqsubseteq e_i$) then $e_1 \sqcup e_2$ is extensional for x, if both are anti-extensional for x ($e_i \sqsubseteq x$) then $e_1 \sqcup e_2 \sqsubseteq x$, anti-extensional for x, and so on.

Similarly, we use the $\ominus$ operation for the extensionality of the meet operation:

$$\frac{\Gamma \vdash e_1 : \mathsf{xb}_1 \quad \Gamma \vdash e_2 : \mathsf{xb}_2}{\Gamma \vdash e_1 \sqcap e_2 : \mathsf{xb}_1 \ominus \mathsf{xb}_2}\ (\text{GLB})$$

The $\ominus$ operation is point-wise, just as the $\oplus$ case. Let the type of e_i be τ, i.e., xb_i are maps from τ-typed variables.

$$\mathsf{x}_\tau \ominus \mathsf{x}'_\tau \;\; = \;\; \{x \mapsto (\mathsf{x}(x) \ominus \mathsf{x}'(x)) \mid x \in \mathit{Var}_\tau\}$$

The commutative $\ominus$ for the extensionality tokens is dual to $\oplus$:

$\ominus$	0	+	-	$\top$
0	0	0	$-$	$-$
+		+	$-$	$\top$
$-$			$-$	$-$
$\top$				$\top$

The rule for lambda expressions is similar to standard typing. The extensionality behaviors of the argument and the result are determined. Note that the result's extensionality (xb_2) can be weaker ($\mathsf{xb}'_2 \sqsubseteq \mathsf{xb}_2$) than that of the body (xb'_2). This relaxation makes the rule safely less restrictive; without it we would have to reject programs in which two functions of varying extensionalities are called in the same application. Lastly, because the actual arguments and the results are independent of the freshly bound parameter, their extensionalities should be independent of the formal parameter:

$$\frac{\Gamma + x : \mathsf{xb}_1 \vdash e : \mathsf{xb}'_2 \quad \mathsf{xb}'_2 \sqsubseteq \mathsf{xb}_2}{\Gamma \vdash \lambda x.e : \mathsf{xb}_1 \to \mathsf{xb}_2} \text{ (LAM)}$$

The rule for recursive function requires that the function name and its body have the same extensionality:

$$\frac{\Gamma \vdash \lambda x.e : \mathsf{xb} \to \mathsf{xb}}{\Gamma \vdash Y\lambda x.e : \mathsf{xb}} \text{ (REC)}$$

The rule for application has nothing particular.

$$\frac{\Gamma \vdash e_1 : \mathsf{xb}_1 \to \mathsf{xb}_2 \quad \Gamma \vdash e_2 : \mathsf{xb}'_1 \quad \mathsf{xb}'_1 \sqsubseteq \mathsf{xb}_1}{\Gamma \vdash e_1\ e_2 : \mathsf{xb}_2} \text{ (APP)}$$

Note that the function's argument extensionality (xb_1) can be weaker ($\mathsf{xb}'_1 \sqsubseteq \mathsf{xb}_1$) than that of the actual argument (xb'_1). This relaxation makes the rule safely less restrictive; without it we would have to reject programs in which the arguments of varying extensionalities are passed to the same function.

The extensionality of the conditional expression should subsume those of the two sub-expressions:

$$\frac{\Gamma \vdash e_3 : \mathsf{xb}_3 \quad \Gamma \vdash e_4 : \mathsf{xb}_4}{\Gamma \vdash \mathit{if}\ e_1 \sqsubseteq e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4 : \mathsf{xb}_3 \sqcup \mathsf{xb}_4} \text{ (IF)}$$

For example, if e_1 is extensional and e_2 is anti-extensional, the result is unknown ($\top$). Contrary to the monotonicity analysis [MY02], careful considerations of the dynamic nature of switching between the two sub-expressions is not critical because, unlike the monotonicity case, both if-branches being extensional is a sufficient condition for the extensionality of the if-expressions. For a better accuracy of our analysis, we might be able to estimate two ranges of each free variable's values that make the condition expression respectively true and false, and analyze each branch only within the corresponding value range. This extra elaboration seems hardly cost-effective for the extensionality analysis hence is not considered here.

In the following examples, for extensionality behavior of type τ, we simply write $\{x_1^{t_1}, \cdots, x_n^{t_n}\}$ to mean

$$\{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\} \cup \{y \mapsto \top \mid y \in \mathit{Var}_\tau \setminus \{x_1, \cdots, x_n\}\}.$$

extensional for x.

Example 2 *Expression $(x \sqcap c_\tau) \sqcup y$ under assumptions $x : \{x^0\}$ and $y : \{y^0\}$ has extensionality $\{x^\top, y^+\}$:*

$$\dfrac{\dfrac{\dfrac{x : \{x^0\}, y : \{y^0\} \vdash x : \{x^0\}\ \textit{(VAR)}}{x : \{x^0\}, y : \{y^0\} \vdash c_\tau : \top_{X_\tau}\ \textit{(CON-3)}}}{x : \{x^0\}, y : \{y^0\} \vdash x \sqcap c_\tau : \{x^0\} \ominus \top_{X_\tau}}\ \textit{(GLB)} \qquad x : \{x^0\}, y : \{y^0\} \vdash y : \{y^0\}}{x : \{x^0\}, y : \{y^0\} \vdash (x \sqcap c) \sqcup y : \{x^\top, y^+\}}\ \textit{(LUB)}$$

The expression's extensionality for x is unknown ($x^\top$). For y, it is from y^0 in assumption to y^+ in result, hence extensional for y.

Example 3 *As an expression where different variables are bound to the same value, consider $(\lambda y.(\lambda f.f\ y)(\lambda x.y))\ z$. Its extensionality under assumption $z : \{z^0\}$ is $\{z^0\}$. That is, the expression's extensionality for z is from z^0 to z^0, hence extensional for z:*

$$\dfrac{\dfrac{z : \{z^0\} \vdash z : \{z^0\} \qquad \dfrac{\dfrac{\dfrac{A}{z : \{z^0\}, y : \{z^0\} \vdash (\lambda f.f\ y)(\lambda x.y) : \{z^0\}}\ \textit{(APP)}}{z : \{z^0\} \vdash (\lambda y.(\lambda f.f\ y)(\lambda x.y)) : \{z^0\} \to \{z^0\}}\ \textit{(LAM)}}{z : \{z^0\} \vdash (\lambda y.(\lambda f.f\ y)(\lambda x.y))\ z : \{z^0\}}\ \textit{(APP)}$$

where sub-proof-trees in A are

$$\dfrac{\dfrac{\dfrac{\vdots}{z : \{z^0\}, y : \{z^0\}, f : \{z^0\} \to \{z^0\} \vdash f\ y : \{z^0\}}\ \textit{(APP)}}{z : \{z^0\}, y : \{z^0\} \vdash \lambda f.f\ y : (\{z^0\} \to \{z^0\}) \to \{z^0\}}\ \textit{(LAM)}$$

and

$$\dfrac{z : \{z^0\}, y : \{z^0\}, x : \{z^0\} \vdash y : \{z^0\}}{z : \{z^0\}, y : \{z^0\} \vdash \lambda x.y : \{z^0\} \to \{z^0\}}\ \textit{(LAM)}$$

Example 4 *As an example that a function is called at different site, consider $(\lambda f.(\lambda z.fy)(fx))(\lambda w_\tau.w)$. Its extensionality under assumption $x : \{x^0\}$ and $y : \{y^0\}$ is $\top_{X_\tau}$, hence the expression's extensionality for x and y are unknown, even though in reality it is extensional for y. This accuracy loss is because our analysis is mono-variant. The smallest functional extensionality for f that allows both "$f\ x$" ($\{x^0\} \to \{x^0\}$) and "$f\ y$" ($\{y^0\} \to \{y^0\}$) is found $\top_{X_\tau} \to \top_{X_\tau}$ by the (LAM) rule.*

4 Soundness

Definition 4 *Let $(s,v) \models \mathsf{xb}$ be the minimal relation between extensionalities xb, values v (lattice elements or functions), and value environments $s \in Var \xrightarrow{fin} Value$ that satisfies:*

$$\begin{array}{rcll}
(s,v) & \models & \mathsf{x}_\tau & \textit{iff} \quad \forall x \in Var_\tau : (s,v) \models x^{\mathsf{x}_\tau(x)} \\
(s,v) & \models & x^+ & \textit{iff} \quad x \in dom(s) \Rightarrow s(x) \sqsubseteq v \\
(s,v) & \models & x^- & \textit{iff} \quad x \in dom(s) \Rightarrow v \sqsubseteq s(x) \\
(s,v) & \models & x^0 & \textit{iff} \quad x \in dom(s) \Rightarrow v = s(x) \\
(s,v) & \models & x^\top & \textit{iff} \quad \textit{true} \\
(s,\lambda x.e) & \models & \mathsf{xb}_1 \to \mathsf{xb}_2 & \textit{iff} \quad ((s,v_1) \models \mathsf{xb}_1) \ \wedge \ (\{v_1/x\}e \hookrightarrow v) \\
 & & & \quad \textit{imply } (s,v) \models \mathsf{xb}_2 \\
(s,Yv) & \models & \mathsf{xb} & \textit{iff} \quad (s,v) \models \mathsf{xb} \to \mathsf{xb}
\end{array}$$

We write $s \models \Gamma$ when the value environment s respects the extensionality assumption Γ:

Definition 5 $s \models \Gamma$ *iff* $dom(\Gamma) = dom(s)$ *nd* $\forall x \in dom(\Gamma) : (s, s(x)) \models \Gamma(x)$.

We write $\{s\}e$ for the expression resulting from substituting s's images for the free variables in e:

Definition 6 $\{s\}e = \{s(x_1)/x_1\}\cdots\{s(x_n)/x_n\}e$ *where* $FV(e) = \{x_1, \cdots, x_n\}$.

Lemma 1 *If* $(s,v) \models \mathsf{xb}$ *and* $\mathsf{xb} \sqsubseteq \mathsf{xb}'$ *then* $(s,v) \models \mathsf{xb}'$.

Proof. Obvious from the definition of $\models$ and the partial order $\sqsubseteq$ in XB. □

Lemma 2 *If* $(s,v) \models \mathsf{xb}_\tau$ *then* $\forall A \subseteq Var_\tau : (s|_A, v) \models \mathsf{xb}_\tau$. *($f|_A$ denotes f restricted to $A \cap dom(f)$.)*

Proof. By the definition of $\models$. □

Finally we are ready to state the correctness result:

Theorem 1 *If* $\Gamma \vdash e : \mathsf{xb}$ *then* $s \models \Gamma$ *and* $\{s\}e \hookrightarrow v$ *imply* $(s,v) \models \mathsf{xb}$.

Proof. We proceed by structural induction on e.

Case $\Gamma \vdash x : \Gamma(x)$.

By definition, $(s, \{s\}x) \models \Gamma(x)$ for $s \models \Gamma$ because $\{s\}x = s(x)$.

Case $\Gamma \vdash \bot_\tau : \{x \mapsto - \mid x \in Var_\tau\}$ or $\Gamma \vdash \top_\tau : \{x \mapsto + \mid x \in Var_\tau\}$.

Obviously, $\bot_\tau \sqsubseteq s(x)$ and $s(x) \sqsubseteq \top_\tau$ for $x \in dom(s)$.

Case for other constants c: $\Gamma \vdash c_\tau : \{x \mapsto \top \mid x \in Var_\tau\}$.

Obviously, $(s,c) \models x^\top$ for $x \in dom(s)$.

Case $\Gamma \vdash \lambda x.e : \mathsf{xb}_1 \to \mathsf{xb}_2$.

By definition, $\Gamma + x : \mathsf{xb}_1 \vdash e : \mathsf{xb}_2'$ and $\mathsf{xb}_2' \sqsubseteq \mathsf{xb}_2$. Let $s \models \Gamma$. By definition $\{s\}\lambda x.e = \lambda x.\{s\}e \hookrightarrow \lambda x.\{s\}e$, because $x \notin dom(s)$. We must show: $(s, \lambda x.\{s\}e) \models \mathsf{xb}_1 \to \mathsf{xb}_2$, that is, $(s,v_1) \models \mathsf{xb}_1$ and $\{v_1/x\}\{s\}e \hookrightarrow v_2$ imply $(s,v_2) \models \mathsf{xb}_2$. From $s \models \Gamma$ and $(s,v_1) \models \mathsf{xb}_1$, $s + x : v_1 \models \Gamma + x : \mathsf{xb}_1$. $\{v_1/x\}\{s\}e \hookrightarrow v_2$ is $\{s + x : v_1\}e \hookrightarrow v_2$. Hence,

by induction hypothesis, $(s + x : v_1, v_2) \models \mathsf{xb}_2'$, which implies $(s + x : v_1, v_2) \models \mathsf{xb}_2$ (by Lemma 1) and in turn $(s, v_2) \models \mathsf{xb}_2$ (by Lemma 2).

Case $\Gamma \vdash Y\lambda x.e : \mathsf{xb}$.
We have to show $(s, Y\lambda x.\{s\}e) \models \mathsf{xb}$ for $s \models \Gamma$. By definition, $\Gamma \vdash \lambda x.e : \mathsf{xb} \to \mathsf{xb}$. By induction hypothesis, $(s, \lambda x.\{s\}e) \models \mathsf{xb} \to \mathsf{xb}$, which, by definition, implies $(s, Y\lambda x.\{s\}e) \models \mathsf{xb}$.

Case $\Gamma \vdash e_1\ e_2 : \mathsf{xb}_2$.
By definition, $\Gamma \vdash e_1 : \mathsf{xb}_1 \to \mathsf{xb}_2$, $\Gamma \vdash e_2 : \mathsf{xb}_1'$, and $\mathsf{xb}_1' \sqsubseteq \mathsf{xb}_1$. Let $s \models \Gamma$ and $(\{s\}e_1)\,(\{s\}e_2) \hookrightarrow v$. We have to show: $(s, v) \models \mathsf{xb}_2$. There are two sub-cases where e_1 evaluates to a non-recursive function or a recursive function.

- When $\{s\}e_1$ evaluates to a non-recursive function, $(\{s\}e_1)\,(\{s\}e_2) \hookrightarrow v$ means

$$\{s\}e_1 \hookrightarrow \lambda x.e, \tag{1}$$
$$\{s\}e_2 \hookrightarrow v', \text{ and} \tag{2}$$
$$\{v'/x\}e \hookrightarrow v. \tag{3}$$

  By induction hypothesis, respectively from (1) and (2),

$$(s, \lambda x.e) \models \mathsf{xb}_1 \to \mathsf{xb}_2 \tag{4}$$

  and $(s, v') \models \mathsf{xb}_1'$ which by Lemma 1 is

$$(s, v') \models \mathsf{xb}_1. \tag{5}$$

  Hence from (3), (4), and (5), $(s, v) \models \mathsf{xb}_2$.

- When e_1 evaluates to a recursive function, $(\{s\}e_1)\,(\{s\}e_2) \hookrightarrow v$ means

$$\{s\}e_1 \hookrightarrow Yv, \tag{6}$$
$$\{s\}e_2 \hookrightarrow v', \tag{7}$$
$$v(Yv) \hookrightarrow \lambda x.e, \text{ and} \tag{8}$$
$$\{v'/x\}e \hookrightarrow v. \tag{9}$$

  By induction hypothesis, respectively from (6) and (7),

$$(s, Yv) \models \mathsf{xb}_1 \to \mathsf{xb}_2 \tag{10}$$

  and $(s, v') \models \mathsf{xb}_1'$ which by Lemma 1 implies

$$(s, v') \models \mathsf{xb}_1. \tag{11}$$

  By definition from (10), $(s, v) \models (\mathsf{xb}_1 \to \mathsf{xb}_2) \to (\mathsf{xb}_1 \to \mathsf{xb}_2)$. Thus from (10) and (8), $(s, \lambda x.e) \models \mathsf{xb}_1 \to \mathsf{xb}_2$, which from (11) and (9) implies $(s, v) \models \mathsf{xb}_2$.

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

5 Algorithm

Our inference system is a variant of type inference system with inclusion constraints [AW93].

Our algorithm consists of three phases: we derive constraints for the extensionality behaviors, simplify the constraints, and then we solve the equations derived from simplified constraints. The conventional fixpoint iteration can be applied to the simplified constraints since every operator ($\oplus, \ominus, f[t/x]$, and $\sqcup$) 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 extensionality that could be inferred in our inference system.

5.1 Extraction of Constraints

Extracted constraint ρ is of the following form:

$$\begin{array}{rrcl} \text{constraint} & \rho & ::= & L \supseteq R \mid \exists\alpha.\rho \mid \rho,\rho \\ \text{left-hand-side term} & L & ::= & \alpha \mid L \to L \\ \text{right-hand-side term} & R & ::= & L \mid \alpha \oplus \alpha \mid \alpha \ominus \alpha \mid \alpha \sqcup \alpha \mid \mathsf{x}_\tau \end{array}$$

where α is a variable for the extensionality behavior, and x_τ is an extensionality behavior of non-functional type τ. The validity of the formula ρ, written as "$\vdash \rho$" is defined as follows. $\{\mathsf{xb}/x\}\rho$ denotes, as usual, the result from substituting xb for variable x in ρ.

$$\frac{\mathsf{xb}_1 \sqsupseteq \mathsf{xb}_2}{\vdash \mathsf{xb}_1 \supseteq \mathsf{xb}_2} \qquad \frac{\vdash \{\mathsf{xb}/\alpha\}\rho}{\vdash \exists\alpha.\rho} \qquad \frac{\vdash \rho_1 \quad \vdash \rho_2}{\vdash \rho_1, \rho_2}$$

We extract the constraints from an expression e using a recursive procedure $C(\Gamma, e, \mathsf{xb})$ (Figure 2). It has a linear time complexity (with respect to the size of e). The size of the generated formula is also linear in e's size.

The constraint generation $C(\Gamma, e, \mathsf{xb})$ and its solution is a correct implementation of our deductive system:

Theorem 2 *If* $\vdash C(\Gamma, e, \mathsf{xb})$ *then* $\Gamma \vdash e : \mathsf{xb}$.

Proof. We proceed by structural induction on e.

Case $\lambda x.e$.

Let $\vdash C(\Gamma, \lambda x.e, \mathsf{xb})$. It implies that there are $\mathsf{xb}_1, \mathsf{xb}_2, \mathsf{xb}_2'$ such that $\vdash C(\Gamma + x : \mathsf{xb}_1, e, \mathsf{xb}_2')$, $\mathsf{xb} \sqsupseteq \mathsf{xb}_1 \to \mathsf{xb}_2$, and $\mathsf{xb}_2' \sqsubseteq \mathsf{xb}_2$. By induction hypothesis, $\Gamma + x : \mathsf{xb}_1 \vdash e : \mathsf{xb}_2'$. By (LAM), $\Gamma \vdash \lambda x.e : \mathsf{xb}_1 \to \mathsf{xb}_2$. Because $\mathsf{xb} \sqsupseteq \mathsf{xb}_1 \to \mathsf{xb}_2$, $\Gamma \vdash \lambda x.e : \mathsf{xb}$.

Case $Y\lambda x.e$.

Let $\vdash C(\Gamma, Y\lambda x.e, \mathsf{xb})$. It implies that $C(\Gamma, \lambda x.e, \mathsf{xb} \to \mathsf{xb})$. By induction hypothesis, $\Gamma \vdash \lambda x.e : \mathsf{xb} \to \mathsf{xb}$. By (REC), $\Gamma \vdash Y\lambda x.e : \mathsf{xb}$.

Case $e_1\ e_2$.

Let $\vdash C(\Gamma, e_1\ e_2, \mathsf{xb})$. It implies $C(\Gamma, e_1, \mathsf{xb}_1 \to \mathsf{xb})$,$C(\Gamma, e_2, \mathsf{xb}_1')$, and $\mathsf{xb}_1' \sqsubseteq \mathsf{xb}_1$. By induction hypothesis, $\Gamma \vdash e_1 : \mathsf{xb}_1 \to \mathsf{xb}$ and $\Gamma \vdash e_2 : \mathsf{xb}_1'$. By (APP), $\Gamma \vdash e_1\ e_2 : \mathsf{xb}$.

Other cases are similar. □

$$
\begin{array}{lcl}
C(\Gamma, \bot_\tau, \mathsf{xb}) & = & \mathsf{xb} \supseteq \{x \mapsto - \mid x \in \mathit{Var}_\tau\} \\
C(\Gamma, c_\tau, \mathsf{xb}) & = & \mathsf{xb} \supseteq \{x \mapsto \top \mid x \in \mathit{Var}_\tau\} \\
C(\Gamma, \top_\tau, \mathsf{xb}) & = & \mathsf{xb} \supseteq \{x \mapsto + \mid x \in \mathit{Var}_\tau\} \\
C(\Gamma, x, \mathsf{xb}) & = & \mathsf{xb} \supseteq \Gamma(x) \\
C(\Gamma, \lambda x.e, \mathsf{xb}) & = & \exists \alpha_1 \alpha_2 \alpha_2'. \\
 & & C(\Gamma + x : \alpha_1, e, \alpha_2'), \\
 & & \mathsf{xb} \supseteq \alpha_1 \to \alpha_2,\ \alpha_2 \supseteq \alpha_2' \\
C(\Gamma, Y\lambda x.e, \mathsf{xb}) & = & C(\Gamma, \lambda x.e, \mathsf{xb} \to \mathsf{xb}) \\
C(\Gamma, e_1\ e_2, \mathsf{xb}) & = & \exists \alpha_1 \alpha_1'. \\
 & & C(\Gamma, e_1, \alpha_1 \to \mathsf{xb}), \quad C(\Gamma, e_2, \alpha_1'), \\
 & & \alpha_1 \supseteq \alpha_1' \\
C(\Gamma, e_1 \sqcup e_2, \mathsf{xb}) & = & \exists \alpha_1 \alpha_2. \\
 & & C(\Gamma, e_1, \alpha_1), \quad C(\Gamma, e_2, \alpha_2), \\
 & & \mathsf{xb} \supseteq \alpha_1 \oplus \alpha_2 \\
C(\Gamma, e_1 \sqcap e_2, \mathsf{xb}) & = & \exists \alpha_1 \alpha_2. \\
 & & C(\Gamma, e_1, \alpha_1), \quad C(\Gamma, e_2, \alpha_2), \\
 & & \mathsf{xb} \supseteq \alpha_1 \ominus \alpha_2 \\
C(\Gamma, \mathit{if}\ e_1 \sqsubseteq e_2\ ?\ e_3\ :\ e_4, \mathsf{xb}) & & \\
 & = & \exists \alpha_1 \alpha_2. \\
 & & C(\Gamma, e_3, \alpha_1), C(\Gamma, e_4, \alpha_2), \\
 & & \mathsf{xb} \supseteq \alpha_1 \sqcup \alpha_2
\end{array}
$$

Figure 2: Constraint Extraction Procedure

5.2 Solving the constraints

Figure 3 shows simplification rules $\mathcal{S}$ for simplifying initial constraints. Rules constits of one transitivity rule and two elimination rules.

We write $A \vdash_{\mathcal{S}} c$ if a constraint c is derivable from A using rules $\mathcal{S}$, and write $\mathcal{S}^*(A)$ for the closure of A under rules $\mathcal{S}$, i.e., the set $\mathit{lfp}(\lambda X.A \cup \{c \mid X \vdash_{\mathcal{S}} c\})$.

An atomic constraint is $\alpha \supseteq \mathit{at}$ whose right-hand-side at (atomic term) explicitly denotes a simple expression of non-functional extensionality behavior x_τ:

$$\mathit{at} \ ::= \ \mathsf{x}_\tau \ \mid \ \alpha \oplus \alpha \ \mid \ \alpha \ominus \alpha$$

Among the constraints $\mathcal{S}^*(A)$, completely dissolved constraints ($\mathit{atom}(\mathcal{S}^*(A))$) constitute the least model of A, where $\mathit{atom}(C) = \{\alpha \supseteq \mathit{at} \in C\}$. We can prove this property easily, because our constraint system is a special case of general setting [AW93].

The simplified atomic constraints constitutes one equation for each unknown; "$\alpha \supseteq \mathit{at}_1, \cdots, \alpha \supseteq \mathit{at}_n$" is equivalent to "$\alpha = \mathit{at}_1 \sqcup \cdots \sqcup \mathit{at}_n$". The least solution of the set of such equations corresponds to the least model of the simplified atomic constraints [CC95].

The least solution is computed by iteration: starting from bottom $\{a \mapsto \bot | a \in \mathit{FVar}_\tau\}$ for every α_τ we repeatedly apply the right-hand-sides of the equations to the intermediate results. This procedure terminates with the least fixed point, because the operators involved ($\oplus, \ominus, \sqcup$) are all monotonic.

$$\frac{L \supseteq \alpha \quad \alpha \supseteq R}{L \supseteq R}\ (\text{TRANS})$$

$$\frac{L_1 \to L_2 \supseteq L_3 \to L_4}{L_3 \supseteq L_1 \quad L_2 \supseteq L_4}\ (\to\text{-ELIM}) \qquad \frac{L \supseteq \alpha_1 \sqcup \alpha_2}{L \supseteq \alpha_1 \quad L \supseteq \alpha_2}\ (\sqcup\text{-ELIM})$$

Figure 3: Simplification Rules $\mathcal{S}$

Example 5 *For* $(\lambda x.(x \sqcap c_\tau) \sqcup y)\ z$*, let* Γ *consist of* $y : \{y^0\}$ *and and* $z : \{z^0\}$*. The constraint generation procedure* $C(\Gamma, (\lambda x.(x \sqcap c_\tau) \sqcup y)\ z, \alpha)$ *returns the following constraint:*

$$\begin{array}{lll} \alpha_1 \supseteq \alpha_2 & \alpha_2 \supseteq z^0 & \alpha_1 \to \alpha \supseteq \alpha_3 \to \alpha_4 \\ \alpha_4 \supseteq \alpha_5 & \alpha_5 \supseteq \alpha_6 \oplus \alpha_7 & \alpha_7 \supseteq y^0 \\ \alpha_6 \supseteq \alpha_8 \ominus \alpha_9 & \alpha_8 \supseteq \alpha_3 & \alpha_9 \supseteq \top_{X_\tau} \end{array}$$

Rules $\mathcal{S}$ *simplifies initial constraints into the simplified atomic constraints:*

$$\begin{array}{lll} \alpha \supseteq \alpha_6 \oplus \alpha_7 & \alpha_1 \supseteq z^0 & \alpha_2 \supseteq z^0 \\ \alpha_3 \supseteq z^0 & \alpha_4 \supseteq \alpha_6 \oplus \alpha_7 & \alpha_5 \supseteq \alpha_6 \oplus \alpha_7 \\ \alpha_6 \supseteq \alpha_8 \ominus \alpha_9 & \alpha_7 \supseteq y^0 & \alpha_8 \supseteq z^0 \\ \alpha_9 \supseteq \top_{X_\tau} & & \end{array}$$

The simplified atomic constraints constitute the equations:

$$\begin{array}{lll} \alpha = \alpha_6 \oplus \alpha_7 & \alpha_1 = z^0 & \alpha_2 = z^0 \\ \alpha_3 = z^0 & \alpha_4 = \alpha_6 \oplus \alpha_7 & \alpha_5 = \alpha_6 \oplus \alpha_7 \\ \alpha_6 = \alpha_8 \ominus \alpha_9 & \alpha_7 = y^0 & \alpha_8 = z^0 \\ \alpha_9 = \top_{X_\tau} & & \end{array}$$

The least solution for α *is computed in 4 iterations, and corresponds to the result of* $(x^0 \ominus \top_{X_\tau}) \oplus y^0$ *which is* $\{x^\top, y^+\}$*.*

Complexity. Let n be the input program's size (the number of sub-expressions), r be its largest type size (the largest number of basic type-components (lattices in our case)[1] in the types of sub-expressions.), and w is the number of variables in the program.

The constraint extraction procedure C takes $O(n)$ time generating $O(n)$ number of constraints. The simplification procedure ($\mathcal{S}^*$) takes the number of times that the simplification rules can be applied. Because the number of possible constraints ($L \supseteq R$) is $n^r \times n^r$ (n^r terms for L and R respectively), and because the simplification rules introduce new constraints that always consist of sub-terms of existing constraints, the simplification procedure reaches the closure within $O(n^{2r})$ in time. After the simplification, the number of equations from the atomic constraints is n. The fixpoint iteration over the equations computes elements of the chains in XB and the chain's length is bounded by $3 \times r \times w$. 3 is the height of the extensionality token domain $T = \{\bot, 0, -, +, \top\}$. Thus we iterate up to $O(r \times w)$ times. Each iteration per equation

[1] E.g. type $(L \to L) \to (L \to L)$ has size 4.

takes $O(r \times w \times n)$ because each equation computes a new extensionality behavior `xb` whose size is $O(r \times w)$, the right-hand-side of each equation can have up to n α's, and constant time is needed for table look-up for each operator. Hence the worst-case time complexity of the fixpoint iteration is $O(n \times (r \times w) \times (r \times w \times n)) = O(n^2 \times w^2 \times r^2)$.

Hence the overall complexity is $O(n^{2r}) + O(n^2 \times w^2 \times r^2)$. In Rabbit programs in reality the largest type size is mostly 2. Hence we expect the complexity in practice is bounded by $O(n^4)$.

6 Conclusion

Our work provides a method of extensionality verification for λ-definable functions over arbitrary finite-height lattices. Static extensionality 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 extensional 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, or from generating extra "joining" operations[LCVH92, LCVH94] necessary to enforce the extensionality of fixpoint iterations. Our work may also suggest a similar solution to existing program analyzer generators like PAG[Mar98].

The extensionality analysis is complementary to the previous work [MY02] on the static monotonicity analysis(checking $\forall x \leq y : f(x) \leq f(y)$). Because the extensionality and monotonicity are incomparable properties and either of them guarantees the termination of the generated analyzers, the two analyses in disjunctive combination will enlarge the set of Rabbit programs accepted by Zoo. Our verification procedure is an inference system, which can be classified as mono-variant flow-insensitive analysis.

We are currently implementing both the extensionality and monotonicity analysis[MY02] inside the Zoo system. Manual experiments are also under way for applying the analyses to existing non-trivial Rabbit programs (e.g. for constant propagation, alias analysis, and exception analyses [YR02, YR97]).

Acknowledgment

We thank Jérôme Feret in many respects. This work started from his comment that the extensionality is another sufficient condition for the termination of the static analyses over finite-height domains. We thank him also for his critical comments on earlier drafts of this paper and for careful listening to continuing questions and intermittent ideas.

References

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 31–41, 1993. 10, 11

[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. 1, 15

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, 1979. 1, 15

[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. 10, 11

[LCVH92] B. Le Charlier and P. Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report TR-CS-92-25, Brown University, Dept. of Computer Science, May 1992. (also as a technical report of Institute of Computer Science, University of Namur). 13

[LCVH94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, January 1994. 13

[Mar98] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998. 13

[MY02] Andrzej Murawski and Kwangkeun Yi. Static monotonicity analysis for lambda-definable functions over lattices. In *The Proceedings of the 3rd International Workshop on Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 139–153, January 2002. 1, 2, 6, 13

[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`. 1, 2, 3, 13

[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-slides.ps`. 1, 13

[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. 13

[YR02] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 277(1-2):185–217, 2002. 13

A Computing A Correct Program Analysis

In the abstract interpretation framework[CC77, CC79], abstract semantic functions are not necessarily monotonic for the correctness of resulting static analyses. Being extensive is another sufficient condition.

Let $(A, \bot, \sqsubseteq)$ and $(\hat{A}, \bot, \sqsubseteq)$ be complete partial order sets with a Galois connection

$$(A, \bot, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\hat{A}, \bot, \sqsubseteq)$$

i.e.

$$\forall x \in A, \hat{x} \in \hat{A} : \alpha(x) \sqsubseteq \hat{x} \Longleftrightarrow x \sqsubseteq \gamma(\hat{x}).$$

Following six facts derived from the above definition are used:

- α is strict.

  Proof. $\alpha(\bot) \sqsubseteq \hat{\bot}$ because $\bot \sqsubseteq \gamma(\hat{\bot})$. □

- $id \sqsubseteq \gamma \circ \alpha$.

  Proof. $\alpha(x) \sqsubseteq \alpha(x)$ and by Galois connection $x \sqsubseteq \gamma(\alpha(x))$. □

- $\alpha \circ \gamma \sqsubseteq id$.

  Proof. $\gamma(\hat{x}) \sqsubseteq \gamma(\hat{x})$ and by Galois connection $\alpha(\gamma(\hat{x})) \sqsubseteq \hat{x}$. □

- γ is monotonic.

  Proof. $\hat{x} \sqsubseteq \hat{y}$ implies $\alpha(\gamma(\hat{x})) \sqsubseteq \hat{y}$, which by Galois connection $\gamma(\hat{x}) \sqsubseteq \gamma(\hat{y})$. □

- α is monotonic.

  Proof. $x \sqsubseteq y$ implies $x \sqsubseteq \gamma(\alpha(y))$, which by Galois connection $\alpha(x) \sqsubseteq \alpha(y)$. □

- α is continuous.

  Proof. We have to show that for any chain S in A, $\alpha(\bigsqcup_{x \in S} x) = \bigsqcup_{x \in S} \alpha(x)$. Because α is monotonic, $\bigsqcup_{x \in S} \alpha(x) \sqsubseteq \alpha(\bigsqcup_{x \in S} x)$. The other direction also holds because $\bigsqcup_{x \in S} x \sqsubseteq \bigsqcup_{x \in S} (\gamma(\alpha(x))) \sqsubseteq \gamma(\bigsqcup_{x \in S} \alpha(x))$ (by monotonicity of γ), hence by Galois connection $\alpha(\bigsqcup_{x \in S} x) \sqsubseteq \bigsqcup_{x \in S} \alpha(x)$. □

Let $F : A \to A$ be a continuous function and $\hat{F} : \hat{A} \to \hat{A}$ be just a function such that

$$\alpha \circ F \sqsubseteq \hat{F} \circ \alpha$$

or dually,

$$F \circ \gamma \sqsubseteq \gamma \circ \hat{F}.$$

The collecting semantics of a program is $lfp F$ which is $\bigsqcup_{i \in \mathbb{N}} (F^i(\bot))$ because F is continuous. The correctness of a program analysis is

$$\alpha(lfp F) \sqsubseteq \text{an iteration algorithm with } \hat{F}.$$

Computing in a finite time an upper bound of

$$\bigsqcup_{i\in\mathbb{N}} \hat{F}^i(\hat{\bot})$$

is such an algorithm.

Why? We know that

$$\forall n \in \mathbb{N} : \alpha \circ F^n \sqsubseteq \hat{F}^n \circ \alpha.$$

Proof.

$$\begin{array}{rcl}
\alpha \circ F^{n+1} & = & \alpha \circ F \circ F^n \\
& \sqsubseteq & \alpha \circ F \circ \gamma \circ \alpha \circ F^n \\
& & (\alpha \circ F \text{ is monotonic and } id \sqsubseteq \gamma \circ \alpha) \\
& \sqsubseteq & \alpha \circ F \circ \gamma \circ \hat{F}^n \circ \alpha \\
& & (\alpha \circ F \circ \gamma \text{ is monotonic and by induction hypothesis}) \\
& \sqsubseteq & \hat{F} \circ \hat{F}^n \circ \alpha. \\
& & (\alpha \circ F \circ \gamma \sqsubseteq \hat{F} \circ \alpha \circ \gamma = \hat{F})
\end{array}$$

□

Thus

$$(\bigsqcup_{i\in\mathbb{N}}(\alpha \circ F^i))(\bot) \sqsubseteq (\bigsqcup_{i\in\mathbb{N}}(\hat{F}^i \circ \alpha))(\bot). \tag{12}$$

The left-hand-side of (12) is

$$\begin{array}{rcl}
(\bigsqcup_{i\in\mathbb{N}}(\alpha \circ F^i))(\bot) & = & \bigsqcup_{i\in\mathbb{N}}(\alpha \circ F^i)(\bot) \\
& & (\text{by def. of the point-wise } \sqcup \text{ for function}) \\
& = & \alpha(\bigsqcup_{i\in\mathbb{N}}(F^i(\bot))) \\
& & (\text{by continuity of } \alpha) \\
& = & \alpha(\mathit{lfp}F). \\
& & (\text{by the least fixpoint theorem})
\end{array}$$

The right-hand-side of (12) is

$$\begin{array}{rcl}
(\bigsqcup_{i\in\mathbb{N}}(\hat{F}^i \circ \alpha))(\bot) & = & \bigsqcup_{i\in\mathbb{N}}((\hat{F}^i \circ \alpha)(\bot)) \\
& & (\text{by def. of the point-wise } \sqcup \text{ for function}) \\
& = & \bigsqcup_{i\in\mathbb{N}}(\hat{F}^i(\hat{\bot})). \\
& & (\text{by strictness of } \alpha)
\end{array}$$

That is, (12) implies

$$\alpha(\mathit{lfp}F) \sqsubseteq \bigsqcup_{i\in\mathbb{N}}(\hat{F}^i(\hat{\bot})).$$

Hence, computing an upper bound of $\bigsqcup_{i\in\mathbb{N}}(\hat{F}^i(\bot))$ is a correct analysis.

Now, how do we compute an upper bound of $\bigsqcup_{i\in\mathbb{N}}(\hat{F}^i(\bot))$? Here we use the monotonicity or the extensionality of $\hat{F}$:

- If $\hat{F}$ is monotonic ($\forall x, y : x \sqsubseteq y \Rightarrow \hat{F}(x) \sqsubseteq \hat{F}(y)$) or extensive ($\forall x : x \sqsubseteq \hat{F}(x)$) over a finite-height domain, then $\bigsqcup_{i\in\mathbb{N}}(\hat{F}^i(\bot)) = \hat{F}^n(\bot)$ for some n because $\{\bot, \hat{F}(\bot), \hat{F}^2(\bot), \cdots\}$ is a chain.

  If $\hat{F}$ is monotonic, such $\hat{F}^n(\bot)$ (a fixpoint of $\hat{F}$) is equal to $lfp\hat{F}$. If $\hat{F}$ is extensive, $\hat{F}^n(\bot)$ is may not be equal to $lfp\hat{F}$.

- If $\hat{F}$ is defined over an infinite-height domain, then computing $\bigsqcup_{i\in\mathbb{N}}(\hat{F}^i(\bot))$ may not terminate. In this case, we compute a finitely increasing chain $\{\hat{F}^i_\nabla(\bot)\}$ such that

$$\bigsqcup_{i\in\mathbb{N}}(\hat{F}^i(\bot)) \sqsubseteq \lim_{i\in\mathbb{N}}(\hat{F}^i_\nabla(\bot))$$

  by defining $\hat{F}^i_\nabla$ (widened $\hat{F}^i$) by a widening operator ∇ as

$$\begin{aligned} \hat{F}^0_\nabla(\bot) &= \hat{F}^0(\bot) \\ \hat{F}^{n+1}_\nabla(\bot) &= \hat{F}^n_\nabla(\bot) \nabla \hat{F}^n(\bot). \end{aligned}$$

  The condition of ∇ for the sequence $\{\hat{F}^i_\nabla(\bot)\}$ to be finitely increasing is:

  $\forall a, b : (a \sqsubseteq a\nabla b) \wedge (b \sqsubseteq a\nabla b)$
  $\forall chain\{x_i\} : chain\ \{y_i\}\ defined\ as\ y_0 = x_0, y_{i+1} = y_i \nabla x_{i+1}\ is\ finite.$