

# On the Power of Abstract Interpretation

Uday S. Reddy, Samuel N. Kamin  
Department of Computer Science  
University of Illinois at Urbana-Champaign  
Urbana, IL 61801  
Net: {reddy,kamin}@cs.uiuc.edu

## Abstract

Increasingly sophisticated applications of static analysis make it important to precisely characterize the power of static analysis techniques. Sekar *et al.* recently studied the power of strictness analysis techniques and showed that strictness analysis is perfect up to variations in constants. We generalize this approach to abstract interpretation in general by defining a notion of *similarity semantics*. This semantics associates to a program a *collection* of interpretations all of which are obtained by blurring the distinctions that a particular static analysis ignores. We define *completeness* with respect to similarity semantics and obtain two completeness results. For first-order languages, abstract interpretation is complete with respect to a standard similarity semantics provided the base abstract domain is linearly ordered. For typed higher-order languages, it is complete with respect to a *logical* similarity semantics again under the condition of linearly ordered base abstract domain.

## 1 Introduction

Static analysis allows compilers to optimize code generation. In recent times, static analysis based on the technique *abstract interpretation* has found increased use [2, 5, 11, 10]. In contrast to traditional applications of static analysis, the newer applications in functional and logic programming implementations achieve radical speedups, sometimes even altering the *complexity* of run time or space. Thus, these applications place an increased burden on the reliability of static analysis. The failure of the analysis to detect some information may mean that the complexity of the algorithm would be altered.

It is therefore necessary to precisely characterize how powerful a static analysis technique is and to state it in terms that the user of a programming language can understand. The latter is necessary because the user of a sophisticated compiler would have to tune his program to fit the compiler's capabilities so that the desired performance can be achieved.

Clearly, no static analysis method can give precise information about all programs. If it did, that would be a violation of Rice's theorem that nontrivial semantics properties of computable functions are undecidable. So, analysis is in general approximate and it is hard to measure how good an approximation it is.

In a recent paper, Sekar *et al.* [13] proposed a new approach to the problem. An abstraction map blurs certain distinctions in the semantic values. Hence, it can never find properties which depend on the distinctions being blurred. However, we may ask whether it finds all the properties which *do not* depend on such distinctions. If it can find all such properties, we can call it *complete*. Sekar *et al.* apply this technique to two particular strictness analysis problems, *viz.*, Mycroft's strictness analysis method for first-order functions [10] and a method of their own formulation [14] which improves on Mycroft's analysis. They show that both these analyses are complete.

In this paper, we follow the approach of Sekar *et al.* and obtain completeness results for a broad *class* of abstract interpretations. Our results show that whenever the base abstract domain is a finite total order, first-order abstract interpretation is complete in

the sense just described. This subsumes the two completeness results proved by Sekar *et al.* and it also covers other abstract interpretations such as the Wadler’s analysis for “list strictness” [16]. Further, our proofs avoid the detailed operational arguments used by Sekar *et al.* and focus on the abstract domain and semantic interpretation.

There are serious impediments to relaxing the condition that the base abstract domain must be a total order. For instance, if product domains are abstracted to products of the component abstract domains (which is not a total order) then abstract interpretation is *not* complete. Thus, some condition is necessary even though the total order restriction may be too strong.

We also generalize the results for *typed higher-order* languages with product and function types. As indicated above, abstract interpretation for products (and function spaces) cannot be complete in the same sense as for the first-order case. So, we propose a revised notion of completeness where structured values such as pairs and functions are not treated as observable values, but have components that are be observable. Two structured values are considered indistinguishable (for the purposes of static analysis) if all their observable components are indistinguishable. This notion of indistinguishability leads to a new notion of completeness. We are able to show that, whenever abstract domains are constructed from totally ordered abstractions for observables, abstract interpretation is complete in that it gives the best information available for all indistinguishable programs.

### 1.1 Relation to previous work

Obviously, the closest work to ours is Sekar *et al.* [13]. We have already stated where we stand with respect to that work: by using denotational arguments instead of operational ones, our results are more general and our proofs simpler.

It is important to understand that this paper does not attempt to offer a new method of strictness analysis. Like Sekar *et al.*, we are only trying to pin down the power of the traditional techniques of Mycroft [10] and Burn, Hankin, and Abramsky [3].

A number of authors have discussed the

optimality of static analyses (e.g. [1, 6, 12]). However, that work does not consider the same problem as concerns us. Where they consider optimality in general — i.e. finding the correct mathematical characterization of optimality — we are looking for a concrete characterization of the information loss in a particular abstract interpretation.

## 2 First order languages

For simplicity, we assume that all functions are  $n$ -ary for some fixed  $n \geq 0$ . Let  $x_1, \dots, x_n$  be a set of  $n$  variables. The language includes a set of constants  $k \in Const$ , a set of primitive functions  $p \in Prim$  and a set of function symbols  $f \in Fun$ . The abstract syntax is

$$\begin{aligned} e &\in \text{Exp} \\ e &::= x_i \mid k \mid p(\bar{e}) \mid f(\bar{e}) \end{aligned}$$

A program is a set of equations of the form

$$f(x_1, \dots, x_n) = e$$

for each  $f$ .

For the semantics, assume a cpo  $\mathcal{D}$  for the value domain. Each constant symbol is interpreted as a value in  $\mathcal{D}$ . Each function symbol is interpreted as a continuous function in  $[\mathcal{D}^n \rightarrow \mathcal{D}]$  and each primitive function symbol is assigned a fixed interpretation. We ambiguously denote the interpretation of a constant  $k$  by  $k$ , and the interpretation of a primitive function symbol  $p$  by  $p$ . Now, the semantics of expressions is determined with respect to an interpretation  $I$  mapping function symbols to functions in  $[\mathcal{D}^n \rightarrow \mathcal{D}]$  and an environment  $\eta$  mapping variables to values in  $\mathcal{D}$ :

$$\begin{aligned} \llbracket x_i \rrbracket I \eta &= \eta(x_i) \\ \llbracket k \rrbracket I \eta &= k \\ \llbracket p(\bar{e}) \rrbracket I \eta &= p(\llbracket \bar{e} \rrbracket I \eta) \\ \llbracket f(\bar{e}) \rrbracket I \eta &= I(f)(\llbracket \bar{e} \rrbracket I \eta) \end{aligned}$$

An interpretation  $I$  that satisfies all the equations of the program is a *model* of the program, and the standard semantics of the program is its least model. As usual, the least model can be expressed as the least fixed point of a functional  $F$  on interpretations. Let  $F(I)$  be the interpretation  $I'$  such that, for all equations  $f(\bar{x}) = e$  in the program,

$$\llbracket f(\bar{x}) \rrbracket I' \eta = \llbracket e \rrbracket I \eta$$

Models of the program are precisely the fixed points of  $F$ . Clearly,  $F$  is continuous and, so, the least fixed point is given by the lub of the Kleene iteration sequence.

### 2.1 Abstract interpretation

For the abstract interpretation, assume a complete lattice  $\mathcal{A}$ . There must be an abstraction function  $abs : \mathcal{D} \rightarrow \mathcal{A}$  that is continuous. The abstraction function induces a corresponding function  $abs : [\mathcal{D}^n \rightarrow \mathcal{D}] \rightarrow [\mathcal{A}^n \rightarrow \mathcal{A}]$  defined by

$$abs(f)(\bar{a}) = \bigsqcup \{abs(f(\bar{v})) \mid abs(\bar{v}) \sqsubseteq \bar{a}\}$$

The abstraction of constant and primitive symbols are  $k^\sharp = abs(k)$  and  $p^\sharp = abs(p)$  respectively. A program can be given semantics in the abstract domain by defining a semantic function  $\llbracket e \rrbracket^\sharp$ . Again, the least abstract model is given by the least fixed point of a continuous function  $F$  on abstract interpretations. The following results relate the standard semantics and the abstract semantics. Whenever  $I$  is an interpretation,  $I^\sharp$  denotes the abstract interpretation given by  $I^\sharp(f) = abs(I(f))$ . Similarly, if  $\eta$  is an environment,  $\eta^\sharp$  denotes the corresponding abstract environment  $\eta^\sharp(x) = abs(\eta(x))$ . The following soundness results are standard [3, 10]:

#### Theorem 2.1 (Soundness)

$$abs(\llbracket e \rrbracket I \eta) \sqsubseteq_{\mathcal{A}} \llbracket e \rrbracket^\sharp I^\sharp \eta^\sharp.$$

**Theorem 2.2 (Soundness)** If  $I$  is the standard semantics of a program and  $I_A$  is its least abstract semantics, then  $I^\sharp \sqsubseteq_{\mathcal{A}} I_A$ .

### 2.2 Similarity semantics

The abstraction map induces an equivalence relation  $\sim$  (called *similarity*) on  $\mathcal{D}$  and  $[\mathcal{D}^n \rightarrow \mathcal{D}]$  as follows. For  $v, w \in \mathcal{D}$ ,

$$v \sim w \Leftrightarrow abs(v) = abs(w)$$

For  $f, g \in [\mathcal{D}^n \rightarrow \mathcal{D}]$ ,

$$f \sim g \Leftrightarrow \forall \bar{v} \in \mathcal{D}^n. f(\bar{v}) \sim g(\bar{v})$$

The idea of similarity is that the abstract interpretation has no way to distinguish between values and functions that are similar. However, the similarity of functions is slightly weaker than having the same abstraction. The notion of similarity can be extended to environments and interpretations point-wise.

**Example** Let  $Bool_\perp$  be the flat cpo of truth values, and  $\mathbf{2} = \{\perp, \top\}$  be the “definedness” abstract domain with the abstraction function  $abs = \{\perp \mapsto \perp, true \mapsto \top, false \mapsto \top\}$ . The only similarity in  $Bool_\perp$  is  $true \sim false$ . Some of the functions in  $[Bool_\perp \rightarrow Bool_\perp]$  are

	$\perp$	$true$	$false$
<i>abort</i>	$\perp$	$\perp$	$\perp$
<i>succ</i>	$\perp$	$true$	$\perp$
<i>nsucc</i>	$\perp$	$false$	$\perp$
<i>fail</i>	$\perp$	$\perp$	$false$
<i>nfail</i>	$\perp$	$\perp$	$true$
<i>id</i>	$\perp$	$true$	$false$
<i>not</i>	$\perp$	$false$	$true$

The function *abort* is not similar to any other function. Among the others,  $succ \sim nsucc$ ,  $fail \sim nfail$  and  $id \sim not$ . Note that all these latter functions have the same abstraction. So, similarity is weaker than having the same abstraction. In Section 4, we strengthen the definition of similarity of functions to include  $succ \sim fail$ . The idea here is that each of them can simulate the other by modifying its input to a similar value. However,  $succ \sim id$  does not hold even for the strengthened definition.

Abstract interpretation of expressions loses information, in general. For instance, whenever  $x \neq \perp$ ,  $succ(fail(x)) = \perp$ , but  $succ^\sharp(fail^\sharp(x)) = \top$ . However, such a loss of information *can* be justified. Since abstract interpretation cannot distinguish between similar values and similar functions, it can only give information that is shared by all the expressions similar to the given expression. Since  $fail \sim nfail$  and  $succ(nfail(x)) = true$ , we cannot expect the abstract interpretation to give precise information about  $succ(fail(x))$ .  $\square$

We formalize the loss of information as follows. We associate with an expression a *class* of interpretations each of which is obtained by confusing the information in the expression via similarity. Now, we can say that an abstraction is *complete* if it identifies the properties shared by all interpretations in the class.

Define a relation  $\sim_{I, \eta} \subseteq (Exp \times \mathcal{D})$  which relates expressions to values by confusing the

values of subexpressions:

$$\begin{array}{ll} x \rightsquigarrow_{I,\eta} v & \text{if } \eta(x) \sim v \\ k \rightsquigarrow_{I,\eta} v & \text{if } k \sim v \\ p(\bar{e}) \rightsquigarrow_{I,\eta} p'(\bar{v}) & \text{if } p \sim p' \wedge \bar{e} \rightsquigarrow_{I,\eta} \bar{v} \\ f(\bar{e}) \rightsquigarrow_{I,\eta} f'(\bar{v}) & \text{if } I(f) \sim f' \wedge \bar{e} \rightsquigarrow_{I,\eta} \bar{v} \end{array}$$

Note that the left hand sides of these rules are *expressions* and the right hand sides are *semantic values*. The  $\rightsquigarrow$  relation is similar to the semantic function except that it allows the values of subexpressions to be modified up to similar values. This gives the following results:

**Lemma 2.3**  $e \rightsquigarrow_{I,\eta} \llbracket e \rrbracket I\eta$ .

Secondly,  $\rightsquigarrow$  is indifferent to similarity perturbations in the inputs and results.

**Lemma 2.4** If  $e \rightsquigarrow_{I,\eta} v$ ,  $I \sim I'$ ,  $\eta \sim \eta'$  and  $v \sim v'$  then  $e \rightsquigarrow_{I',\eta'} v'$ .

An interpretation  $I$  is said to be a *similarity-model* of a program if, for each equation  $f(\bar{x}) = e$  in the program,

$$e \rightsquigarrow_{I,\eta} \llbracket f(\bar{x}) \rrbracket I\eta$$

However, unlike for the standard semantics, similarity-models cannot be expressed as fixed points of functionals. We use an alternative formulation to achieve the same effect. Given two interpretations  $I$  and  $J$  of a program  $P$ , we say that  $I$  improves to  $J$  ( $I \hookrightarrow_P J$ ) iff, for every equation  $f(\bar{x}) = e$ ,

$$e \rightsquigarrow_{I,\eta} \llbracket f(\bar{x}) \rrbracket J\eta$$

Like the  $\rightsquigarrow_{I,\eta}$  relation,  $\hookrightarrow_P$  is indifferent to similarity perturbations:

**Lemma 2.5** If  $I' \sim I$ ,  $I \hookrightarrow_P J$  and  $J \sim J'$  then  $I' \hookrightarrow_P J'$ .

*Proof:* The only use made of  $I$  in  $I \hookrightarrow_P J$  is in the condition  $e \rightsquigarrow_{I,\eta} \llbracket f(\bar{x}) \rrbracket J\eta$  which is indifferent to a similarity perturbation in  $I$ . Similarly, the only use made of  $J$  is in interpreting  $f(\bar{x})$ . Since  $\llbracket f(\bar{x}) \rrbracket J\eta = J(f)(\eta(\bar{x})) \sim J'(f)(\eta(\bar{x}))$ , we have that  $I' \hookrightarrow_P J'$ .<sup>1</sup>  $\square$

Note that  $I$  is a similarity-model of  $P$  iff  $I \hookrightarrow_P I$ . Further:

<sup>1</sup>Even though  $\llbracket f(\bar{x}) \rrbracket J\eta \sim \llbracket f(\bar{x}) \rrbracket J'\eta$ , it is important to note that this does not generalize to arbitrary expressions in place of  $f(\bar{x})$ . For example,  $f(g(x))$  has dissimilar values in the two similar interpretations  $J = \{f \mapsto \text{succ}, g \mapsto \text{fail}\}$  and  $J' = \{f \mapsto \text{succ}, g \mapsto \text{nfail}\}$ .

### Lemma 2.6

Every model of  $P$  is a similarity-model of  $P$ .

*Proof:* If  $I$  is a model of  $P$  then, for every equation  $f(\bar{x}) = e$ ,  $\llbracket f(\bar{x}) \rrbracket I\eta = \llbracket e \rrbracket I\eta$ . By Lemma 2.3, we have  $e \rightsquigarrow_{I,\eta} \llbracket f(\bar{x}) \rrbracket I\eta$ . Hence,  $I \hookrightarrow_P I$ .  $\square$

Intuitively, a similarity-model allows constants and primitives to be “confused”. As a trivial special case, it may perform no confusion at all, in which case it is a model.

The predicate  $e \rightsquigarrow_{I,\eta} v$  is inclusive in  $I$ ,  $\eta$  and  $v$ , i.e., given increasing sequences  $\{I_i\}_i$ ,  $\{\eta_i\}_i$  and  $\{v_i\}_i$  such that  $e \rightsquigarrow_{I_i,\eta_i} v_i$ , we have  $e \rightsquigarrow_{\sqcup_i I_i, \sqcup_i \eta_i} \sqcup_i v_i$ . This is clear from the fact that the values  $v$  have a continuous dependence on  $\eta$  and  $I$ . Similarly, the relation  $\hookrightarrow_P$  is also inclusive. We can now give a Kleene-like construction for similarity-models:

**Theorem 2.7** Let  $P$  be a program and  $I_\perp = I_0 \hookrightarrow_P I_1 \hookrightarrow_P I_2 \hookrightarrow_P \dots$  be an increasing sequence of interpretations.  $\sqcup_i I_i$  is a similarity-model of  $P$ .

*Proof:* Consider the subsequence  $I_1 \hookrightarrow_P I_2 \hookrightarrow_P \dots$ . The given sequence is related to this by  $\hookrightarrow_P$  relation. Hence, by inclusivity,  $\sqcup_{i \geq 0} I_i \hookrightarrow_P \sqcup_{i \geq 1} I_i$ . But, the two lubs are equal. Hence  $\sqcup_i I_i \hookrightarrow_P \sqcup_i I_i$  showing that it is a similarity-model.  $\square$

We call similarity-models expressible as lubs of such sequences *reachable similarity-models*. They capture the notion of confusing the standard semantics of programs. Other unreachable similarity-models are of no computational interest.

## 3 Completeness

For the purpose of this paper, we impose two conditions on the abstractions. First,  $\mathcal{A}$  must be a finite *linearly ordered* partial order. (Such a partial order is automatically a complete lattice). Second, the abstraction map  $\text{abs} : \mathcal{D} \rightarrow \mathcal{A}$  should be *upward-faithful*, i.e., whenever  $\text{abs}(v) \sqsubseteq_{\mathcal{A}} \text{abs}(w)$ , there must exist a  $w' \in \mathcal{D}$  such that  $\text{abs}(w') = \text{abs}(w)$  and  $v \sqsubseteq_{\mathcal{D}} w'$ . This essentially means that  $a \sqsubseteq_{\mathcal{A}} b$  iff the inverse images of  $a$  and  $b$  are related by the Hoare power domain order. See [3] for a discussion of the use of Hoare power domains in abstract interpretation. Upward-faithfulness implies:

**Lemma 3.1** There is a subdomain  $\mathcal{C} \subseteq \mathcal{D}$  isomorphic to  $\mathcal{A}$ .

*Proof:* We proceed to build  $\mathcal{C}$  by induction on the structure of  $\mathcal{A}$ . First,  $\perp_{\mathcal{D}} \in \mathcal{C}$ . Now consider any  $a \in \mathcal{A}$  such that for all  $a' \sqsubseteq a$ , there exists  $x \in \mathcal{C}$  such that  $\text{abs}(x) = a'$ , but for which there is no  $x \in \mathcal{C}$  with  $\text{abs}(x) = a$ . It follows immediately from upward-faithfulness that there is an  $x \in \text{abs}^{-1}(a)$  such that for all  $a' \sqsubseteq a$  there are  $y \in \text{abs}(a')$  with  $y \sqsubseteq x$ . Add this  $x$  to  $\mathcal{C}$ . Proceed until all elements of  $\mathcal{A}$  (and thus of  $\mathcal{D}$ ) are accounted for. ■

We will also denote by  $\mathcal{C}$  the function (a retract of  $\mathcal{D}$ , in technical terms)  $\mathcal{C}(y)$  is the element  $x$  of  $\mathcal{C}$  such that  $\text{abs}(y) = \text{abs}(x)$ .

**Lemma 3.2** If  $f$  is an arbitrary function, and  $\text{abs} \circ f$  is continuous, then  $\mathcal{C} \circ f$  is continuous.

Upward-faithfulness generalizes to functions, environments and interpretations in a natural way:

**Lemma 3.3** Let  $\text{abs} : \mathcal{D} \rightarrow \mathcal{A}$  be an upward-faithful abstraction function.

1. For  $f, g \in \mathcal{D}^n \rightarrow \mathcal{D}$ , if  $\text{abs}(f) \sqsubseteq \text{abs}(g)$  then there exists  $g'$  such that  $\text{abs}(g') = \text{abs}(g)$  and  $f \sqsubseteq g'$ .
2. For environments  $\eta_1$  and  $\eta_2$ , if  $\eta_1^\sharp \sqsubseteq \eta_2^\sharp$  then there exists  $\eta_2'$  such that  $(\eta_2')^\sharp = \eta_2^\sharp$  and  $\eta_1 \sqsubseteq \eta_2'$ .
3. For interpretations  $I$  and  $J$ , if  $I^\sharp \sqsubseteq J^\sharp$  then there exists  $J'$  such that  $(J')^\sharp = J^\sharp$  and  $I \sqsubseteq J'$ .

The conditions we have imposed on  $\mathcal{A}$  allow us to simplify the definition of  $\text{abs}$  for functions:

**Lemma 3.4** For any  $f \in \mathcal{D}^n \rightarrow \mathcal{D}$  and  $\bar{a} \in \mathcal{A}^n$  such that all the components of  $\bar{a}$  are in the range of  $\text{abs}$ ,

$$\text{abs}(f)(\bar{a}) = \bigsqcup \{ \text{abs}(f(\bar{v})) \mid \text{abs}(\bar{v}) = \bar{a} \}.$$

*Proof:* Let  $\bar{w}$  be such that  $\text{abs}(\bar{w}) = \bar{a}$ . By definition,

$$\begin{aligned} \text{abs}(f)(\bar{a}) &= \text{abs}(f)(\text{abs}(\bar{w})) \\ &= \bigsqcup \{ \text{abs}(f(\bar{u})) \mid \text{abs}(\bar{u}) \sqsubseteq \text{abs}(\bar{w}) \}. \end{aligned}$$

Upward-faithfulness implies that this equals:

$$\bigsqcup \{ \text{abs}(f(\bar{u})) \mid \exists \bar{v}. \text{abs}(\bar{v}) = \text{abs}(\bar{w}) \wedge \bar{u} \sqsubseteq \bar{v} \},$$

which, by monotonicity of  $\text{abs}$ , equals:

$$\bigsqcup \{ \text{abs}(f(\bar{v})) \mid \text{abs}(\bar{v}) = \text{abs}(\bar{w}) \},$$

as required.

**Lemma 3.5** For all  $f$  and  $\bar{v}$ , there exists a  $\bar{u}$  such that  $\text{abs}(\bar{v}) = \text{abs}(\bar{u})$  and  $\text{abs}(f)(\text{abs}(\bar{v})) = \text{abs}(f(\bar{u}))$ .

*Proof:* This follows from the previous lemma, together with the fact that  $\mathcal{A}$  is finite and linearly-ordered, which implies that  $\bigsqcup$  is just the max operation.

Now, the following result strengthens Theorem 2.1 by replacing  $\sqsubseteq_{\mathcal{A}}$  with equality.

**Theorem 3.6 (Completeness)** Let  $I$  be an interpretation and  $\eta$  an environment. For every expression  $e$ , there is a  $v \in \mathcal{D}$  such that  $e \rightsquigarrow_{I, \eta} v$  and  $\text{abs}(v) = \llbracket e \rrbracket^\sharp I^\sharp \eta^\sharp$ .

*Proof:* By structural induction on  $e$ .

- $\llbracket x \rrbracket^\sharp I^\sharp \eta^\sharp = \eta^\sharp(x) = \text{abs}(\eta(x))$ . Since  $x \rightsquigarrow_{I, \eta} \eta(x)$ , the conclusion is obtained.
- $\llbracket k \rrbracket^\sharp I^\sharp \eta^\sharp = k^\sharp = \text{abs}(k)$ . Since  $k \rightsquigarrow_{I, \eta} k$ , the conclusion is obtained.
- $\llbracket p(\bar{e}) \rrbracket^\sharp I^\sharp \eta^\sharp = p^\sharp(\llbracket \bar{e} \rrbracket^\sharp I^\sharp \eta^\sharp)$ . By inductive hypothesis, there exist  $\bar{v}$  such that  $\bar{e} \rightsquigarrow_{I, \eta} \bar{v}$  and  $\text{abs}(\bar{v}) = \llbracket \bar{e} \rrbracket^\sharp I^\sharp \eta^\sharp$ . Now,  $p^\sharp(\text{abs}(\bar{v})) = \text{abs}(p(\bar{u}))$  for some  $\bar{u} \sim \bar{v}$ , by lemma 3.5. Since  $\bar{e} \rightsquigarrow_{I, \eta} \bar{v}$  and  $\bar{v} \sim \bar{u}$ , we also have  $\bar{e} \rightsquigarrow_{I, \eta} \bar{u}$ . Hence, the conclusion follows.
- $\llbracket f(\bar{e}) \rrbracket^\sharp I^\sharp \eta^\sharp = I^\sharp(f)(\llbracket \bar{e} \rrbracket^\sharp I^\sharp \eta^\sharp)$ . This case is similar to the above because  $I^\sharp(f) = \text{abs}(I(f))$ .

□

This result is somewhat stronger than what is needed. It says that, among the interpretations associated with an expression, there is one that precisely realizes the information given by the abstract interpretation. Hence, it follows that the abstract interpretation gives the information shared by all the interpretations of the expression.

### Theorem 3.7 (Completeness for programs)

If  $J_A$  is the least abstract interpretation of a program  $P$ , there is a reachable similarity-model  $I$  of  $P$  such that  $I^\sharp = J_A$ .

*Proof:* Let  $F^\sharp$  be the program's functional on abstract interpretations, i.e., for each equation  $f(\bar{x}) = e$ ,  $F^\sharp(J)(f)(\bar{w}) = \llbracket e \rrbracket^\sharp J\{\bar{x} \mapsto \bar{w}\}$ .  $J_A$  is the least fixed point of  $F^\sharp$ . Hence, it is the lub of the increasing sequence  $J_i = (F^\sharp)^i(J_\perp)$ . We exhibit an increasing sequence  $I_0 \hookrightarrow_P I_1 \hookrightarrow_P \dots$  such that  $(I_i)^\sharp = J_i$ . Furthermore, for all  $i$  and  $f$ ,  $I_i(f) = \mathcal{C} \circ I_i(f)$ .

- Let  $I_0$  be the undefined interpretation. Clearly,  $(I_0)^\sharp = J_0$  and  $I_0(f) = \mathcal{C} \circ I_0(f)$ .
- Suppose  $(I_i)^\sharp = J_i$ . Let  $f$  be a function defined by  $f(\bar{x}) = e$  and  $\eta$  the environment  $\{\bar{x} \mapsto \bar{w}\}$  for some  $\bar{w} \in \mathcal{D}^n$ . Define  $\hat{f}(\bar{w})$  to be some  $v$  (whose existence is guaranteed by 3.6) such that  $e \rightsquigarrow_{I_i, \eta} v$  and  $abs(v) = \llbracket e \rrbracket^\sharp I_i \eta^\sharp$ . Now let  $I_{i+1}(f) = \mathcal{C} \circ \hat{f}$ . We have, for all  $\bar{w}$ ,  $abs(I_{i+1}(f)(\bar{w})) = \llbracket e \rrbracket^\sharp J_i\{\bar{x} \mapsto \bar{w}\} = J_{i+1}(f)(abs(\bar{w}))$ , so that  $I_{i+1}^\sharp = J_{i+1}$ . Finally,  $I_i^\sharp \sqsubseteq I_{i+1}^\sharp$  implies that  $I_i = \mathcal{C} \circ I_i \sqsubseteq \mathcal{C} \circ I_{i+1} = I_{i+1}$ .

■

This result strengthens Theorem 2.2 by showing the existence of an interpretation  $I$  such that  $I^\sharp$  is equal to the abstract semantics. The interpretation  $I$  is obtained by confusing the standard semantics of the program. Since all such confusions are indistinguishable in the abstraction, this is indeed the best that can be achieved by abstract interpretation.

Note that the sequence  $J_i$  actually converges in finite time, say at element  $J_n$ , so that in fact  $I_n^\sharp = J_A$ . This suggests the following:

**Corollary 3.8** If  $J_A$  is the least abstract interpretation of program  $P$ , there is a program  $P'$  obtained from  $P$  by a finite number of unfoldings of function definitions, followed by replacements of constants by similar constants, such that  $\llbracket P' \rrbracket^\sharp = J_A$ .

### 3.1 Discussion

To get a flavor of the completeness result, consider the program

$$f(x) = \mathbf{if} \ x = 0 \ \mathbf{then} \ 1 \ \mathbf{else} \ \perp$$

Given the abstraction  $abs : Nat_\perp \rightarrow \mathbf{2}$  which maps  $\perp_{Nat}$  to  $\perp$  and all other values to  $\top$ , the abstraction of  $f$  is

$$f^\sharp(x) = \begin{cases} \perp, & \text{if } x = \perp \\ \top, & \text{otherwise} \end{cases}$$

There is loss of information involved in this abstraction because, for instance,  $f(1) = \perp$  which abstracts to  $\perp$  whereas  $f^\sharp(abs(1)) = \top$ . However, the result  $f(1) = \perp$  depends on the fact that  $(1 = 0) = false$ . Since the abstraction map has been defined to blur the distinction between *true* and *false*, the abstract interpretation of  $f(1)$  does not have the information that  $(1 = 0) = false$ . It would have to produce the same results even if  $(1 = 0)$  were *true*. Our similarity semantics of programs is defined in such a way that it blurs all the distinctions that are lost in the abstraction. Both  $1 = 0 \rightsquigarrow false$  and  $1 = 0 \rightsquigarrow true$  are possible. Hence, both  $f(1) \rightsquigarrow_{I, \eta} \perp$  as well as  $f(1) \rightsquigarrow_{I, \eta} 1$  hold and, so, the result of abstract interpretation denotes the best possible information.

The main condition used in this completeness result is that the abstract domain must be linearly ordered. This may appear to be a severe restriction, but some of the oft-used abstract interpretations are based on linearly ordered abstract domains. The strictness analysis of Mycroft [11, 10] uses the two point definedness domain  $\mathbf{2} = \{\perp \sqsubseteq \top\}$ . Sekar *et. al.* [13] work with concrete domains of terms and analyze for three properties:  $\perp$  denoting undefined,  $H$  denoting definedness of the head (outermost constructor symbol), and  $\top$  denoting complete definedness. This analysis can be modeled by the abstract domain  $\{\perp \sqsubseteq H \sqsubseteq \top\}$  which is linearly ordered. Wadler [16] uses a four-point abstract domain for analyzing list strictness which is also linearly ordered. Our results show that all these analyses are complete in that they provide the best information representable in these abstract domains.

It seems that the linear order restriction can be relaxed. What is needed is some kind

of condition stating that the lubs of abstract values are “precise”. That is, when we take the lub of two abstract values for lack of certainty about which of two computation paths would be followed, the lub should precisely capture this uncertainty without introducing any extra loss of information. However, we are not able to formalize this intuition.

To give a flavor of loss of completeness that arises with imprecise lubs, consider the concrete domain  $Nat_{\perp} \times Nat_{\perp}$  abstracted to  $\mathbf{2} \times \mathbf{2}$ . The pair  $(\top, \top)$  is the lub of  $(\perp, \top)$  and  $(\top, \perp)$ . Consider the program

$$f(x) = \mathbf{if } x = 0 \mathbf{ then } (1, \perp) \mathbf{ else } (\perp, 2)$$

Its abstraction is

$$f^{\sharp}(x) = \begin{cases} (\perp, \perp) & \text{if } x = \perp \\ (\top, \top) & \text{otherwise} \end{cases}$$

If  $x \neq \perp$ ,  $f(x)$  is either  $(1, \perp)$  or  $(\perp, 2)$  neither of which realizes the analysis  $(\top, \top)$ . More precisely, there is no  $v$  such that  $f(x) \rightsquigarrow_{I, \eta} v$  and  $abs(v) = (\top, \top)$ . Theorem 3.6 fails. Thus, analysis of product domains cannot be complete in the sense of this section. In the next section, we adopt these notions to typed lambda calculus. It would then be seen that analysis of product domains can be complete in a different sense.

## 4 Typed lambda calculus

In this section, we extend the results of Section 2 to typed lambda calculus with products and function spaces. The language includes a set of base types  $\beta$  and a set of constants  $k^{\tau}$  (which subsume primitive functions). The abstract syntax is

$$\begin{aligned} \tau &\in \text{Type} \\ \tau &::= \beta \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \\ e^{\tau} &\in \text{Exp}_{\tau} \\ e^{\tau} &::= x^{\tau} \mid k^{\tau} \mid e^{\tau \times \tau'} . 1 \mid e^{\tau' \times \tau} . 2 \\ &\quad \mid e_1^{\tau'} \rightarrow^{\tau} e_2^{\tau'} \mid \mathbf{fix } e^{\tau \rightarrow \tau} \\ e^{\tau_1 \times \tau_2} &::= \langle e_1^{\tau_1}, e_2^{\tau_2} \rangle \\ e^{\tau \rightarrow \tau'} &::= \lambda x^{\tau} . e^{\tau'} \end{aligned}$$

To be precise, one would have to define the context-sensitive syntax in terms of type inference rules as in [4, 9]. We often drop type superscripts because they can be automatically reconstructed.

For the semantics, assume a cpo  $\mathcal{D}_{\beta}$  for each base type  $\beta$ .  $\mathcal{D}_{\tau_1 \times \tau_2}$  is the cpo-product

$\mathcal{D}_{\tau_1} \times \mathcal{D}_{\tau_2}$  and  $\mathcal{D}_{\tau_1 \rightarrow \tau_2}$  is the continuous function space  $[\mathcal{D}_{\tau_1} \rightarrow \mathcal{D}_{\tau_2}]$ . The assignment  $\mathcal{D}$  of domains to types is called a *typed lambda structure*. The semantic functions are conventional:

$$\begin{aligned} \llbracket x \rrbracket \eta &= \eta(x) \\ \llbracket k \rrbracket \eta &= k \\ \llbracket \langle e_1, e_2 \rangle \rrbracket \eta &= \langle \llbracket e_1 \rrbracket \eta, \llbracket e_2 \rrbracket \eta \rangle \\ \llbracket e . 1 \rrbracket \eta &= \mathit{fst}(\llbracket e \rrbracket \eta) \\ \llbracket e . 2 \rrbracket \eta &= \mathit{snd}(\llbracket e \rrbracket \eta) \\ \llbracket \lambda x . e \rrbracket \eta &= \lambda v . \llbracket e \rrbracket \eta[x \mapsto v] \\ \llbracket e_1 e_2 \rrbracket \eta &= (\llbracket e_1 \rrbracket \eta)(\llbracket e_2 \rrbracket \eta) \\ \llbracket \mathbf{fix } e \rrbracket \eta &= \mathit{fix}(\llbracket e \rrbracket \eta) \end{aligned}$$

For the abstract semantics, choose a similar typed lambda structure  $\mathcal{A}$  with the restriction that each  $\mathcal{A}_{\beta}$  is finite and linearly ordered. The abstraction function  $abs_{\beta} : \mathcal{D}_{\beta} \rightarrow \mathcal{A}_{\beta}$  should be strict, continuous and upward-faithful. It is extended to other types by:

$$\begin{aligned} abs_{\tau_1 \times \tau_2}(\langle v_1, v_2 \rangle) &= \langle abs_{\tau_1}(v_1), abs_{\tau_2}(v_2) \rangle \\ abs_{\tau_1 \rightarrow \tau_2}(f) &= \\ &\lambda a . \bigsqcup \{ abs_{\tau_2}(f(v)) \mid abs(v) \sqsubseteq_{\mathcal{A}} a \} \end{aligned}$$

Note that  $abs_{\tau}$  is continuous for each  $\tau$ . The abstract semantic function is denoted by  $\llbracket e \rrbracket^{\sharp}$ .

Theorem 2.1 relating the standard and abstract semantics generalizes to the typed lambda calculus [3].

### Theorem 4.1 (Soundness)

$$abs_{\tau}(\llbracket e^{\tau} \rrbracket \eta) \sqsubseteq \llbracket e^{\tau} \rrbracket^{\sharp} \eta^{\sharp}.$$

This subsumes Theorem 2.2 by the presence of *fix* terms.

### 4.1 Logical similarity semantics

The notion of similarity is now extended to typed lambda structures by defining  $\sim_{\tau}$  for each type  $\tau$  as follows:

- $v \sim_{\beta} w$  iff  $abs_{\beta}(v) = abs_{\beta}(w)$ .
- $p \sim_{\tau_1 \times \tau_2} q$  iff  $\mathit{fst}(p) \sim_{\tau_1} \mathit{fst}(q)$  and  $\mathit{snd}(p) \sim_{\tau_2} \mathit{snd}(q)$ .
- $f \sim_{\tau_1 \rightarrow \tau_2} g$  iff  $\forall v \in \mathcal{D}_{\tau_1} . (\exists w \sim_{\tau_1} v . f(v) \sim_{\tau_2} g(w))$

Note that the symmetry of  $\sim$  implies the converse  $g(w) \sim_{\tau_2} f(v)$  is also included in the last condition. In contrast to the treatment

in Section 3.1, pairs are treated here as unobservable structures. So, two pairs are similar iff all their observable components are similar. Two functions are similar if, for every input, there is a similar input so that the two functions produce similar results for the two similar inputs. For example, among the functions in  $[Bool_{\perp} \rightarrow Bool_{\perp}]$  mentioned in section 2.3,  $succ \sim nsucc \sim fail \sim nfail$  and  $id \sim not$ . So, this notion of similarity is stronger than that used in section 2 for the first-order case. However, the results of that section hold for the stronger notion of similarity as well because similarity interpretations of terms allow similarity perturbations in function inputs.

Define a relation  $\rightsquigarrow_{\tau, \eta} \subseteq (Exp_{\tau} \times \mathcal{D}_{\tau})$  for recursion-free terms as in Figure 1. The last two conditions denote the extra loss of information for structured values. Since these two conditions are similar to the notion of a *logical relation* [8, 15], we call this “logical” similarity semantics.

For recursive terms, we need some definitions. Let  $F \subseteq [\mathcal{D}_{\tau} \rightarrow \mathcal{D}_{\tau}]$  be a similarity-closed set of functions, i.e.,  $f \in F$  and  $f' \sim f$  implies  $f' \in F$ . Define the improvement relation  $\hookrightarrow_F \subseteq (\mathcal{D}_{\tau} \times \mathcal{D}_{\tau})$  as  $v \hookrightarrow_F w$  iff  $w = f(v)$  for some  $f \in F$ . A *similarity-fixpoint* of  $F$  is a value  $v$  such that  $v \hookrightarrow_F v$ .

**Lemma 4.2**  $\hookrightarrow_F$  is inclusive, i.e., whenever  $\{v_i\}$  and  $\{w_i\}$  are increasing sequences such that  $v_i \hookrightarrow_F w_i$ ,  $\bigsqcup_i v_i \hookrightarrow_F \bigsqcup_i w_i$ .

*Proof:* Since the abstract domains are finite, there exists a finite  $k$  such that  $v_k \sim v_i$  and  $w_k \sim w_i$  for all  $i \geq k$ . Let  $f_k \in F$  be the function such that  $w_k = f_k(v_k)$ . Since similarity is closed under lubs of sequences (*abs* is continuous),  $v_k \sim \bigsqcup_i v_i$  and  $w_k \sim \bigsqcup_i w_i$ . Hence, there is a function  $f \in F$  such that  $\bigsqcup_i w_i = f(\bigsqcup_i v_i)$ . ■

**Theorem 4.3** Let  $\perp_{\tau} = v_0 \hookrightarrow_F v_1 \hookrightarrow_F \dots$  be an increasing improvement sequence. Then  $\bigsqcup_i v_i$  is a similarity-fixpoint of  $F$ .

*Proof:* Consider the subsequence  $v_1 \hookrightarrow_F v_2 \hookrightarrow_F \dots$ . The given sequence is related to this by the  $\hookrightarrow_F$  relation. Since  $\hookrightarrow_F$  is inclusive,  $\bigsqcup_{i \geq 0} v_i \hookrightarrow_F \bigsqcup_{i \geq 1} v_i$ . The two lubs being equal, we have  $\bigsqcup_i v_i \hookrightarrow_F \bigsqcup_i v_i$ . ■

As in the first-order case, we call the similarity-fixpoints expressible by lubs of

such sequences *reachable*. Now, the similarity semantics of  $fix\ e$  is specified as follows:  $fix\ e \rightsquigarrow_{\tau, \eta} v$  if  $v$  is a reachable similarity-fixpoint of  $\{f \mid e \rightsquigarrow_{\tau, \eta} f\}$ .

The basic properties of the  $\rightsquigarrow$  relation generalize from the first-order case:

**Lemma 4.4**  $e \rightsquigarrow_{\eta} \llbracket e \rrbracket_{\eta}$ .

**Lemma 4.5** If  $e \rightsquigarrow_{\eta} v$ ,  $\eta \sim \eta'$  and  $v \sim v'$  then  $e \rightsquigarrow_{\eta'} v'$ .

**Lemma 4.6** The relation  $e \rightsquigarrow_{\eta} v$  is inclusive in  $\eta$  and  $v$ , i.e., if  $\{\eta_i\}$  and  $\{v_i\}$  are increasing sequences of environments and values respectively such that  $e \rightsquigarrow_{\eta_i} v_i$  then  $e \rightsquigarrow_{\bigsqcup_i \eta_i} \bigsqcup_i v_i$ .

*Proof:* Denote the lubs of  $\{\eta_i\}$  and  $\{v_i\}$  by  $\eta^{\infty}$  and  $v^{\infty}$  respectively. The proof is by structural induction on  $e$ . We show selected cases:

- If  $x \rightsquigarrow_{\eta_i} v_i$  then  $v_i \sim \eta_i(x)$ . By inductive hypothesis,  $v^{\infty} \sim (\eta^{\infty})(x)$ . Hence, the conclusion follows.
- If  $\langle e, e' \rangle \rightsquigarrow_{\eta_i} \langle v_i, w_i \rangle$  then  $e \rightsquigarrow_{\eta_i} v_i$  and  $e' \rightsquigarrow_{\eta_i} w_i$ . By inductive hypothesis,  $e \rightsquigarrow_{\eta^{\infty}} v^{\infty}$  and  $e' \rightsquigarrow_{\eta^{\infty}} w^{\infty}$ . Hence, the conclusion follows.
- If  $\lambda x.e \rightsquigarrow_{\eta_i} f_i$  then, for all  $v$   $e \rightsquigarrow_{\eta_i[x \mapsto v]} f_i(v)$ . By inductive hypothesis,  $e \rightsquigarrow_{\eta^{\infty}[x \mapsto v]} f^{\infty}(v)$ . Hence,  $\lambda x.e \rightsquigarrow_{\eta^{\infty}} f^{\infty}$ .

■

## 4.2 Completeness

**Theorem 4.7** For all expressions  $e^{\tau}$  and environments  $\eta$ ,

$$\llbracket e \rrbracket^{\sharp} \eta^{\sharp} = \bigsqcup \{abs_{\tau}(v) \mid e \rightsquigarrow_{\tau, \eta} v\}$$

*Proof:* Let  $S$  be the set of abstract values on the right. Since  $\llbracket e \rrbracket_{\eta} \in S$ , clearly  $\bigsqcup S \subseteq \llbracket e \rrbracket^{\sharp} \eta^{\sharp}$ . To show  $\llbracket e \rrbracket^{\sharp} \eta^{\sharp} \subseteq \bigsqcup S$ , we show that there exists  $v$  such that  $e \rightsquigarrow_{\tau, \eta} v$  and  $abs_{\tau}(v) = \llbracket e \rrbracket^{\sharp} \eta^{\sharp}$ . Proceed by induction on  $e$ . (We drop the type subscripts  $\tau$  in the following).

- If  $e = x$ ,  $\llbracket x \rrbracket^{\sharp} \eta^{\sharp} = abs(\eta(x))$ . Since  $x \rightsquigarrow_{\eta} \eta(x)$ , choose  $\eta(x)$  for  $v$ .

$x$	$\rightsquigarrow_{\tau, \eta}$	$v$	if $\eta(x) \sim v$
$k$	$\rightsquigarrow_{\tau, \eta}$	$v$	if $k \sim v$
$\langle e_1, e_2 \rangle$	$\rightsquigarrow_{\tau_1 \times \tau_2, \eta}$	$\langle v_1, v_2 \rangle$	if $e_1 \rightsquigarrow_{\tau_1, \eta} v_1$ and $e_2 \rightsquigarrow_{\tau_2, \eta} v_2$
$e.1$	$\rightsquigarrow_{\tau_1, \eta}$	$\text{fst}(p)$	if $e \rightsquigarrow_{\tau_1 \times \tau_2, \eta} p$
$e.2$	$\rightsquigarrow_{\tau_2, \eta}$	$\text{snd}(p)$	if $e \rightsquigarrow_{\tau_1 \times \tau_2, \eta} p$
$\lambda x. e$	$\rightsquigarrow_{\tau_1 \rightarrow \tau_2, \eta}$	$f$	if $\forall v. e \rightsquigarrow_{\tau_2, \eta}[x \rightarrow v] f(v)$
$e_1 e_2$	$\rightsquigarrow_{\tau_2, \eta}$	$f(v)$	if $e_1 \rightsquigarrow_{\tau_1 \rightarrow \tau_2, \eta} f$ and $e_2 \rightsquigarrow_{\tau_1, \eta} v$

Further, for all expressions  $e$  of appropriate types:

$e$	$\rightsquigarrow_{\tau_1 \times \tau_2, \eta}$	$\langle v_1, v_2 \rangle$	if $e.1 \rightsquigarrow_{\tau_1, \eta} v_1$ and $e.2 \rightsquigarrow_{\tau_2, \eta} v_2$
$e$	$\rightsquigarrow_{\tau_1 \rightarrow \tau_2, \eta}$	$f$	if for all $e', e' \rightsquigarrow_{\tau_1, \eta} v \Rightarrow ee' \rightsquigarrow_{\tau_1, \eta} f(v)$

Figure 1: Logical similarity semantics—Definition

- If  $e = \lambda x. e'$ , we need to exhibit  $f$  such that  $\lambda x. e' \rightsquigarrow_{\eta} f$  and  $\text{abs}(f) = \llbracket \lambda x. e' \rrbracket^{\sharp} \eta^{\sharp} = \lambda a. \llbracket e' \rrbracket^{\sharp} \eta^{\sharp}[x \rightarrow a]$ . Let  $w$  be an arbitrary value (of the domain type of  $f$ ). By inductive hypothesis, there exists  $v$  such that  $e' \rightsquigarrow_{\eta[x \rightarrow w]} v$  and  $\text{abs}(v) = \llbracket e' \rrbracket^{\sharp} \eta^{\sharp}[x \rightarrow a]$ . By choosing appropriate  $v$ 's for  $w$ 's, we can form a function  $f$  satisfying the condition.
- If  $e = e_1 e_2$ , by inductive hypothesis, we have  $v_1$  and  $v_2$  such that  $e_i \rightsquigarrow_{\eta} v_i$  and  $\text{abs}(v_i) = \llbracket e_i \rrbracket^{\sharp} \eta^{\sharp}$ . By the “logical” condition of the definition of  $\rightsquigarrow$ ,  $e_1 \rightsquigarrow_{\eta} v_1$  means that for all  $e_1 e_2 \rightsquigarrow_{\eta} v_1(v_2)$ . Also, we have  $\text{abs}(v_1(v_2)) = \text{abs}(v_1)(\text{abs}(v_2)) = \llbracket e_1 e_2 \rrbracket^{\sharp} \eta^{\sharp}$ . So, choose  $v_1(v_2)$  to be  $v$ .
- The cases of  $\langle e_1, e_2 \rangle$ ,  $e'.1$  and  $e'.2$  can be handled similarly.
- If  $e = \text{fix } e'$ , by inductive hypothesis, we have  $f$  such that  $e' \rightsquigarrow_{\eta} f$  and  $\text{abs}(f) = \llbracket e' \rrbracket^{\sharp} \eta^{\sharp}$ . Call the latter  $f^{\sharp}$ . Let  $F$  be the similarity closure of  $\{f\}$ , i.e.,  $F$  is the least set satisfying  $f' \sim f \Rightarrow f' \in F$ . Consider the sequences

$$\begin{aligned} v_0 \hookrightarrow_F v_1 \hookrightarrow_F v_2 \dots &\subseteq \mathcal{D} \\ a_0 \sqsubseteq a_1 \sqsubseteq a_2 \dots &\subseteq \mathcal{A} \end{aligned}$$

where  $v_0 = \perp_{\mathcal{D}}$ ,  $a_0 = \perp_{\mathcal{A}}$  and  $a_{i+1} = f^{\sharp}(a_i)$ . Using the definitions of  $\rightsquigarrow$  and  $\text{abs}$ , we have that  $\text{abs}(v_i) = a_i$ . By continuity of  $\text{abs}$ ,  $\text{abs}(\bigsqcup_i v_i) = \bigsqcup_i a_i$ . The former is a reachable similarity fixed

point of  $F$  and so,  $\text{fix } e \rightsquigarrow_{\eta} \bigsqcup_i v_i$ . The latter is the least fixed point of  $f^{\sharp}$ , and, so, equal to  $\llbracket \text{fix } e \rrbracket^{\sharp} \eta^{\sharp}$ . So, choose  $\bigsqcup_i v_i$  to be  $v$ .

■

## 5 Conclusion

We provide here a general framework for discussing the completeness issues of static analysis using abstract interpretation. The essential idea is that abstract interpretation blurs certain distinctions in the concrete values by mapping them to abstract values. If the concrete values mapped to identical abstract values are considered indistinguishable, we can raise the question whether the analysis can detect all the properties shared by indistinguishable classes of programs.

Our results give an affirmative answer for two cases: For the analysis of first-order function definitions using a totally ordered abstract domain, the abstract interpretation is complete. For higher-order functional programs with pairs, the abstract interpretation is complete if structured values considered indistinguishable whenever their components are indistinguishable.

We also show that the results do not generalize to arbitrarily ordered abstract domains. Essentially, the results fail because the least upper bounds of abstract values can involve loss of information. It should be possible to formalize such loss of information in a precise way and duplicate our results for that case. This may be able to cover, for example, the

abstraction of sum domains by products of abstract domains.

Forward abstract interpretation cannot give certain essential information about program evaluation such as the property of head-strictness [7, 17]. The analysis for such properties can be characterized as the abstract interpretation of the continuation semantics of programs. It would be interesting to check if such backward analysis has completeness properties of the kind investigated here.

## References

- [1] ABRAMSKY, S. Abstract interpretation, logical relations, and kan extensions. *J. of Logic and Computation* 1, 1 (Jul 1990), 5–40.
- [2] ABRAMSKY, S., AND HANKIN, C. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.
- [3] BURN, G., HANKIN, C., AND ABRAMSKY, S. The theory of strictness analysis for higher order functions. In *LNCS 217*, Springer-Verlag, Berlin, 1985, pp. 42–62.
- [4] DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *ACM Symp. on Princ. of Programming Languages* (1982), pp. 207–212.
- [5] DEBRAY, S. K. Static inference of modes and data dependencies in logic programs. *ACM Trans. Program. Lang. Syst.* 11, 3 (July 1989), 418–450.
- [6] HUNT, S. *PERs Generalise Projections for Strictness Analysis*. Research Report DOC 90/14, Imperial College, London, Oct 1990.
- [7] KAMIN, S. N. Head-strictness is not a monotonic abstract property. *Information Processing Letters* (1992), (to appear).
- [8] MITCHELL, J. C. Type systems for programming languages. In *Handbook of Theoretical Computer Science, Volume B*, J. van Leeuwen, Ed., North-Holland, Amsterdam, 1990, pp. 365–458.
- [9] MITCHELL, J. C., AND HARPER, R. The essence of ML. In *ACM Symp. on Princ. of Programming Languages* (1988), ACM, pp. 28–46.
- [10] MYCROFT, A. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, Univ. of Edinburgh, Dec. 1981.
- [11] MYCROFT, A. The theory and practice of transforming call-by-need into call-by-value. In *4th Intl. symp. on Programming*, Springer-Verlag, 1980, pp. 269–281.
- [12] NIELSON, F. Expected forms of data flow analyses. In *Programs as Data Objects*, H. Ganzinger and N. D. Jones, Eds., Springer-Verlag, Berlin, 1985, pp. 172–191. (LNCS Vol. 217).
- [13] SEKAR, R. C., MISHRA, P., AND RAMAKRISHNAN, I. V. On the power and limitation of strictness analysis based on abstract interpretation. In *ACM Symp. on Princ. of Programming Languages* (Jan 1991), ACM, pp. 37–48.
- [14] SEKAR, R. C., PAWAGI, S., AND RAMAKRISHNAN, I. V. Small domains spell fast strictness analysis. In *ACM Symp. on Princ. of Programming Languages* (Jan 1990), ACM, pp. 169–183.
- [15] STATMAN, R. Logical relations and the typed lambda calculus. *Information and Control* 65 (1985), 85–97.
- [16] WADLER, P. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds., Ellis Horwood, 1987, p. .
- [17] WADLER, P., AND HUGHES, R. J. M. Projections for strictness analysis. In *Conf. on Functional Program. Lang. and Comput. Arch.*, G. Kahn, Ed., Springer-Verlag, Berlin, 1987, pp. 385–407.