Strictness Analysis in Logical Form

Thomas P. Jensen*
Imperial College†

Abstract

This paper presents a framework for comparing two strictness analysis techniques: Abstract interpretation and non–standard type inference. The comparison is based on the representation of a lattice by its ideals. A formal system for deducing inclusions between ideals of a lattice is presented and proved sound and complete. Viewing the ideals as *strictness properties* we use the formal system to define a program logic for deducing strictness properties of expressions in a typed lambda calculus. This *strictness logic* is shown to be sound and complete with respect to the abstract interpretation, which establishes the main result that strictness analysis by type–inference and by abstract interpretation are equally powerful techniques.

1 Introduction

Abstract interpretation is a well–established technique for static analysis of programs. Its virtue is its strong connection with denotational semantics which provides a means of proving the analysis correct. Its vice is that the process of doing abstract interpretation is quite costly in terms of computation time, although developments in the area of algorithms for finding fixpoints have improved the situation substantially [7]. Recently, a number of researchers have suggested that analyses be based on non–standard type systems. Examples include Wadler's linear types [14] and Kuo and Mishra's strictness types [11]. The advantage of this approach is that there exist efficient algorithms for checking and inferring types of programs. However, the correctness of these analyses has received rather less attention than the implementation aspects and to the best of our knowledge there has not been given a satisfactory explanation of what can be achieved by abstract interpretation but not by the type system and *vice versa*.

The purpose of this paper is to present a framework for relating abstract interpretations to non–standard type systems. We will concentrate on strictness analysis in this paper but the framework is applicable to other analyses as well. The basic idea is to view these non–standard type systems as program logics. The relationship between abstract interpretation and non–standard type inference can then be seen as a special case of the relationship between denotational and axiomatic semantics. This topic has been investigated by a number of researchers and we shall be following a line similar to that of [1].

*This work was partially supported by ESPRIT grant BRA 3124 SEMANTIQUE

†Author's address: Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, U.K. Email: tpj@doc.ic.ac.uk

We begin with a section in which we describe the kind of properties we shall be using in the strictness logic. Essentially these properties are those that are given by downwards closed subsets of the strictness domains. We prove the important fact that our abstract domains can equally well be presented as an ordered set of elements or as a collection of properties of the domain, because these presentations determine each other. We then give a formal system that axiomatises the entailment relation between the strictness properties and we prove that the formal system is sound and complete with respect to the interpretation of properties as subsets of an ordered set. Section 4 reviews strictness analysis by abstract interpretation and introduces a program logic for deducing strictness properties of programs. We give a proof of the main theorem, that the strictness logic can prove exactly those strictness properties that we could find by abstract interpretation. Section 5 concludes with relating the present work to that of others and sketching some directions for future developments.

2 The ideal representation of lattices

This section presents some lattice–theoretic facts that will form the basis for our comparison between abstract interpretation and program logics. We discuss what a property over an abstract domain is and show how the order structure on the domain can be used to single out certain properties of interest. We shall see that a lattice–theoretic result concerning the representation of a semilattice by its ideals can be interpreted as saying that an abstract domain is fully determined by a certain kind of properties definable over the domain together with the logical entailment relation associated with these properties.

In this paper we shall take an abstract interpretation to be a denotational semantics (see *e.g.*, [13]) for which the denotation of any program is effectively computable. One way of achieving effectiveness is by requiring the domains used in the interpretation to be finite. This on the other hand introduces a loss of information that in general makes it impossible to calculate the condition in a branching expression. The interpretation of a branching expression is therefore usually taken to be a combination of the interpretations of the branches. The ability to combine denotations is ensured by requiring the domain to have binary joins. A least element, $\perp$, is added for the interpretation of fixpoint operators. We summarise this in a definition.

Definition 1 *A* join–semilattice *(or $\sqcup$–semilattice) is an ordered set $(S, \sqsubseteq)$ with a binary join operator $\sqcup : S \times S \rightarrow S$ and a distinguished element $\perp \in S$ such that*

- *$a \sqcup b$ is the least upper bound of a and b in S.*
- *$\perp$ is the least element in $(S, \sqsubseteq)$.*

The category of all finite join–semilattices with order–preserving maps as morphisms is denoted by $\mathbf{JSL}_f$. For the rest of the paper we shall take our domains to be objects in $\mathbf{JSL}_f$. We note that a finite join–semilattice actually is a complete lattice, but we shall not use this fact in this section.

Similarly a meet–semilattice is an ordered set with a greatest lower bound operator, $\wedge$, and a greatest element, $\top$. The category of finite meet–semilattices is denoted by $\mathbf{MSL}_f$.

In ordinary set–based logic a property can be identified with the subset of elements satisfying the property, and, conversely, any subset determines a property. Property P then entails property Q if the set of elements satisfying P is contained in the set of elements satisfying Q. With an ordering imposed on the set of elements it is possible to restrict attention to a smaller set of properties by only studying subsets satisfying certain constraints related to the order structure. *E.g.* with a two–point domain $\{\bot, \top\}$ with $\bot$ denoting non–termination and $\top$ denoting termination, we initially have properties $\{\bot\}$, $\{\top\}$ and $\{\bot, \top\}$ where the latter property corresponds to the "empty" property satisfied by all elements. If we are not interested in whether a program definitely terminates, *i.e.*, satisfies property $\{\top\}$, we can require our propositions to be subsets downwards closed with respect to the ordering on the domain. This will give us the two properties $\{\bot\}$ and $\{\bot, \top\}$ with $\{\bot\}$ entailing $\{\bot, \top\}$. This is exactly the kind of properties on which we shall build a program logic in the following sections. We formalise this notion of property using the concept of an *ideal* of a semilattice.

Definition 2 *An* ideal *of a join–semilattice S is a non–empty subset $I \subseteq S$ satisfying*

- *I is down–closed,* i.e., *$x \in I$ and $x \sqsubseteq y$ implies $y \in I$.*
- *I is closed under binary joins* i.e., *$x, y \in I$ implies $x \sqcup y \in I$.*

An ideal is principal *if it is the down–closure of a single element a. In this case we write I as $\downarrow(a)$*

What structure do we have on the collection of ideals? The union of two ideals is in general not an ideal but the intersection is. Furthermore the intersection of two ideals A and B is the greatest ideal contained in both A and B. There is also an ideal that contains all other ideals of S namely S itself. In other words the set of ideals of S forms a meet–semilattice with set intersection as the meet operation. If S is finite so is the set of ideals of S. We thus have a mapping $\mathsf{Idl} : \mathbf{JSL}_f \rightarrow \mathbf{MSL}_f$ which maps a join–semilattice to its meet–semilattice of ideals.

Under the assumption that a join–semilattice S is finite we can say more. As all ideals are finite and closed under finite joins every ideal must contain a maximal element, hence all ideals are principal and the mapping $a \rightarrow \downarrow(a)$ is a bijective correspondence between the elements of the join–semilattice S and the meet–semilattice $\mathsf{Idl}(S)$.

So far we have taken the domain as starting point and constructed a collection of properties over the domain. The next question is: Is this process reversible? Can we reconstruct the original join–semilattice from the meet–semilattice of properties? The answer is yes and we show this by giving a general method for constructing a domain from a set of properties we want to be definable over the domain. The key idea here is that *an element should be fully described by the properties it satisfies.* Following this idea we construct a new lattice, the elements of which are sets of properties. However, only sets satisfying certain consistency requirements should be considered. The sets must be closed under implication and conjunction, because if property P_1 holds of an element p and P_1 implies P_2, then P_2 will hold of p as well. Similarly, if properties P_1 and P_2 hold of an element p then so does $P_1 \wedge P_2$. This is formalised by the notion of a filter of a meet–semilattice.

Definition 3 *A* filter *of a meet–semilattice M is a non–empty subset $F \subseteq M$ that satisfies*

- *F is upwards closed.*
- *$P_1, P_2 \in F \Rightarrow P_1 \wedge P_2 \in F$.*

A filter is called *principal* if it is the upwards closure of a single element. Note that a filter of a meet–semilattice M turns into an ideal if the ordering on M is reversed. This means that many of the results about ideals in join–semilattices can be translated into results about filters in meet–semilattices "by duality". Especially it is easy to see that the filters of a finite meet–semilattice are upwards closures of single elements (*i.e.*, they are principal filters).

We take filters of properties to be the elements of a domain. The ordering on the elements is determined by the set of properties in the filter. The key observation here is that *the more properties an element satisfies, the less defined it is.* In particular the least defined element is the element satisfying all properties. So we order the filters by reverse inclusion and the least upper bound operation then becomes set intersection. This gives a mapping $\mathsf{Fil} : \mathbf{MSL}_f \rightarrow \mathbf{JSL}_f$ that sends a meet–semilattice to its join–semilattice of filters.

How does a join–semilattice L relate to the join–semilattice $\mathsf{Fil}(\mathsf{Idl}(L))$ obtained from the closed subsets of L? There is an obvious embedding $\eta : L \rightarrow \mathsf{Fil}(\mathsf{Idl}(L))$ defined by

$$\eta(a) = \{I \in \mathsf{Idl}(L) \mid a \in I\}.$$

This maps a to the set of ideals containing a, or in logical terms, to the set of properties satisfied by a.

Proposition 1 *The mapping $\eta : L \rightarrow \mathsf{Fil}(\mathsf{Idl}(L))$ is a $\mathbf{JSL}_f$–isomorphism*

Proof. First observe that the set $\{I \in \mathsf{Idl}(L) \mid a \in I\}$ is equal to the set of ideals containing the ideal $\downarrow(a)$. In other words $\eta(a) = \uparrow(\downarrow(a))$, so $\eta(a)$ is a (principal) filter. On the other hand, every filter is principal because of finiteness, so every filter is of the form $\uparrow(I)$ for some ideal I, and every ideal is principal so $I = \downarrow(a)$ for some a, so every filter is of the form $\uparrow(\downarrow(a))$, ie η is surjective. Injectivity follows from the fact that for two different elements, $a_1 \neq a_2$ of L there exists an ideal containing one but not the other, hence $\eta(a_1) \neq \eta(a_2)$. Finally we must show that η is order–preserving:

$$\begin{aligned} a_1 \sqsubseteq a_2 &\Rightarrow \downarrow(a_1) \sqsubseteq \downarrow(a_2) \\ &\Rightarrow \uparrow(\downarrow(a_1)) \sqsubseteq \uparrow(\downarrow(a_2)) \\ &\Rightarrow \eta(a_1) \sqsubseteq \eta(a_2) \qquad \blacksquare \end{aligned}$$

The conclusion of this is that our abstract domains can equally well be presented as a join–semilattice of elements or as a meet–semilattice of properties in the sense that these two structures determines each other up to isomorphism.

3 Strictness domains presented axiomatically

The next step is to apply the domain representation defined in the previous section to the domains encountered in abstract interpretation used for strictness analysis. The strictness analysis employs a set of types defined by the grammar:

$$\sigma ::= \mathbf{2} \mid \sigma_1 \to \sigma_2$$

To each type is associated a lattice as defined by the function $D : \mathbf{Types} \to \mathbf{JSL}_f$

$$D(\mathbf{2}) = \{0, 1\} \text{ with } 0 \sqsubseteq 1$$

$$D(\sigma_1 \to \sigma_2) = D(\sigma_1) \to_m D(\sigma_2)$$

where the symbol $\to_m$ denotes the space of order–preserving (monotone) maps.

From the previous section we have the representation of any $D(\sigma)$ by its meet–semilattice of ideals. However, we do not have a proper syntactic description of $\mathsf{Idl}(D(\sigma))$. The purpose of this section is to provide an axiomatic system that precisely defines the meet–semilattice of ideals for all $D(\sigma)$. For every type σ we define a logical theory

$$\mathcal{L}(\sigma) = (L(\sigma), \wedge, \leq, =)$$

where $L(\sigma)$ is a set of formulae (denoting properties), the binary operator $\wedge$ is conjunction, $\leq$ corresponds to entailment and $=$ is logical equivalence between formulae. We shall see that the ordered set of propositions $\mathcal{L}(\sigma)$ quotiented by logical equivalence is order–isomorphic to the meet–semilattice of ideals of $D(\sigma)$, thus providing a purely axiomatic description of $\mathsf{Idl}(D(\sigma))$.

There are several ways of axiomatising meet–semilattices. The most concise presentation would formalise arbitrary, finite (including empty) meets. We have chosen to introduce some redundancy by explicitly naming certain elements in the meet–semilattices of ideals. We do this to make the presentation easier to read and to obtain a formal system close to that of [11]. *E.g.* the top element, $\mathtt{t}$, in every meet–semilattice is in effect the empty meet, so by axiomatising the top element separately we only have to define binary meets to obtain all finite meets. Similarly we introduce a formula for the least ideal, $\mathtt{f} = \{\perp\}$. Of course the price we have to pay is that we have to add as extra axioms all facts provable about these formulae in the original system.

The sets of formulae are defined inductively as follows:

- $\mathtt{t}, \mathtt{f} \in L(\sigma)$
- $\dfrac{\phi, \psi \in L(\sigma)}{\phi \wedge \psi \in L(\sigma)}$
- $\dfrac{\phi \in L(\sigma), \psi \in L(\tau)}{\phi \to \psi \in L(\sigma \to \tau)}$

The following axioms and rules define the semilattice structure on each $L(\sigma)$.

- $\varphi \leq \varphi$
- $\dfrac{\varphi \leq \psi, \psi \leq \chi}{\varphi \leq \chi}$
- $\dfrac{\varphi \leq \psi_1, \varphi \leq \psi_2}{\varphi \leq \psi_1 \wedge \psi_2}$
- $\varphi \wedge \psi \leq \varphi$
- $\varphi \wedge \psi \leq \psi$
- $\dfrac{\varphi \leq \psi, \psi \leq \varphi}{\varphi = \psi}$
- $\dfrac{\varphi = \psi}{\varphi \leq \psi, \psi \leq \varphi}$
- $\varphi \leq \mathtt{t}$
- $\mathtt{t}_{\sigma \rightarrow \tau} \leq \mathtt{t}_\sigma \rightarrow \mathtt{t}_\tau$
- $\mathtt{f} \leq \varphi$
- $\mathtt{t}_\sigma \rightarrow \mathtt{f}_\tau \leq \mathtt{f}_{\sigma \rightarrow \tau}$
- $\varphi \rightarrow \psi_1 \wedge \varphi \rightarrow \psi_2 \leq \varphi \rightarrow (\psi_1 \wedge \psi_2)$
- $\dfrac{\varphi' \leq \varphi, \psi \leq \psi'}{\varphi \rightarrow \psi \leq \varphi' \rightarrow \psi'}$

The connection between this logical system defining $\mathcal{L}(\sigma)$ and the collection of ideals of $D(\sigma)$ is established by means of semantic functions $[\![\_]\!]_\sigma : \mathcal{L}(\sigma) \rightarrow \mathsf{Idl}(D(\sigma))$:

$$
\begin{array}{lll}
[\![\mathtt{t}]\!]_\sigma & = & D(\sigma) \\
[\![\mathtt{f}]\!]_\sigma & = & \{\perp_{D(\sigma)}\} \\
[\![\bigwedge_{i \in I} \psi_i]\!]_\sigma & = & \bigcap_{i \in I} [\![\psi_i]\!]_\sigma \\
[\![\varphi \rightarrow \psi]\!]_{\sigma \rightarrow \tau} & = & \{f \in D(\sigma \rightarrow \tau) \mid f([\![\varphi]\!]) \subseteq [\![\psi]\!]\}
\end{array}
$$

We now set out to give a characterisation of the connection between the ideals in the domains and the formal system defined above. The following theorems state that all ideals of a domain are indeed definable in the logical system and that the logic is sound and complete, *i.e.*, that all inclusions that hold between ideals in the domains can be proved in the logic (completeness) and no inclusions but those that hold can be proved (soundness).

Before we embark on proving the theorems we need some results about how functions on lattices can be approximated by simpler objects. For these results it is important that for finite join–semilattices it is possible to define a greatest lower bound operator, *i.e.*, they are in reality complete lattices. It is well known [1] from Domain Theory that all continuous functions between two Scott-domains can be expressed using some simple functions called *step-functions*. These step functions are used to approximate functions from below. We shall be concerned with approximating functions (on lattices) from above, hence we study the dual to a step function:

Definition 4 *Let A, B be lattices and assume that $(a, b) \in A \times B$. The* co–step function *$\lfloor (a,b) \rfloor \colon A \rightarrow B$ is defined by*

$$
\lfloor (a,b) \rfloor (x) = \begin{cases} b & \textit{if } x \sqsubseteq a \\ \top & \textit{otherwise} \end{cases}
$$

We shall also use this notation for the greatest lower bound of a set of co–step functions. Let $u = \{(a_i, b_i) \mid i \in I\}$ be a set of pairs drawn from $A \times B$. The function $\lfloor u \rfloor$: A $\to$B induced by u is defined by

$$\lfloor u \rfloor(a) = \sqcap\{b_i \mid a \sqsubseteq a_i\}$$

Any function is the greatest lower bound of the co–step functions greater than function itself, so this gives a way of using co–step functions to build other functions. In fact we can be a bit more economical with the set of approximants due to the following lemma:

Lemma 1 *Let* $f : A \to B$ *Then*

$$f = \lfloor \{(a, f(a)) \mid a \in A\} \rfloor = \sqcap\{\lfloor (a, f(a)) \rfloor \mid a \in A\}$$

The usefulness of co–step functions is due to the fact that the principal ideal generated by the co–step function $\lfloor$(a,b)$\rfloor$ is the set of functions that map ideal $\downarrow$(a) into the ideal $\downarrow$(b). We generalise this fact in the following lemma:

Lemma 2 *Let* $u = \{(a_i, b_i) \mid i \in I\}$*. Then*

$$\downarrow(\lfloor u \rfloor) = \cap_{i \in I}\{f : A \to B \mid f(\downarrow(a_i)) \subseteq \downarrow(b_i)\}$$

Proof: $f \sqsubseteq \lfloor u \rfloor$

$$\begin{aligned}
&\Leftrightarrow \forall a \in A. f(a) \sqsubseteq \sqcap\{b_i \mid a \sqsubseteq a_i\} \\
&\Leftrightarrow \forall i \in I. \forall a \in A. a \sqsubseteq a_i \Rightarrow f(a) \sqsubseteq b_i \\
&\Leftrightarrow \forall i \in I. f \sqsubseteq \lfloor (a_i, b_i) \rfloor \\
&\Leftrightarrow f \in \cap_{i \in I}\{\downarrow(\lfloor (a_i, b_i) \rfloor)\} \\
&\Leftrightarrow f \in \cap_{i \in I}\{f : A \to B \mid f(\downarrow(a_i)) \subseteq \downarrow(b_i)\} \quad \blacksquare
\end{aligned}$$

The following theorem states that all ideals can be denoted by a formula from our formal system. In the formulation of the theorem we use the fact that all ideals are principal *i.e.*, downwards closures of a single element.

Theorem 1 (Definability) *For all elements* $a : D(\sigma)$ *there exist a* $\varphi_a \in \mathcal{L}(\sigma)$ *such that*

$$\downarrow(a) = [\![\varphi_a]\!]_\sigma$$

Proof. The proof works by structural induction over the type σ. For the base type **2** it is straightforward to see that the two elements 0 and 1 corresponds to the formulae `f` and `t`, respectively. Now, let $f \in D(A \to B)$. Then we have:

$$\begin{aligned}
\downarrow(f) &= \downarrow(\lfloor \{(a, f(a)) \mid a \in A\} \rfloor) && \text{by lemma 1} \\
&= \cap_{a \in A}\{g : A \to B \mid g(\downarrow(a)) \subseteq \downarrow(f(a))\} && \text{by lemma 2} \\
&= [\![\textstyle\bigwedge_{a \in A} \varphi_a \to \varphi_{f(a)}]\!] && \text{where } \varphi_a \text{ and } \varphi_{f(a)} \text{ are given by the ind. hyp.} \quad \blacksquare
\end{aligned}$$

Theorem 2 (Soundness) *For all elements $\varphi, \psi \in \mathcal{L}(\sigma)$:*

$$\varphi \leq \psi \quad \Rightarrow \quad [\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma$$

Proof. The proof amounts to check that each axiom and rule is sound, which can be done using simple set-theoretical arguments. ■

Theorem 3 (Completeness) *For all elements $\varphi, \psi \in \mathcal{L}(\sigma)$:*

$$[\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma \quad \Rightarrow \quad \varphi \leq \psi$$

Proof. Induction over the structure of σ. For the base case **2** we can easily prove the only non-trivial inclusion $[\![\mathtt{f}]\!] \subseteq [\![\mathtt{t}]\!]$ using either the rule $\varphi \leq \mathtt{t}$ or $\mathtt{f} \leq \varphi$. For the inductive step we let $\bigwedge_{i \in I}(\varphi_i \to \psi_i)$ and $\bigwedge_{j \in J}(\varphi_j \to \psi_j)$ be two elements of $\mathcal{L}(\sigma \to \tau)$ and let a_n, b_n be such that $\downarrow(a_n) = [\![\varphi_n]\!]$ and $\downarrow(b_n) = [\![\psi_n]\!]$. Dropping subscripts on the conjunctions we have:

$$\begin{array}{ll}
& [\![\bigwedge(\varphi_i \to \psi_i)]\!] \subseteq [\![\bigwedge(\varphi_j \to \psi_j)]\!] \\
\Leftrightarrow & \bigcap [\![\varphi_i \to \psi_i]\!] \subseteq \bigcap [\![\varphi_j \to \psi_j]\!] \\
\Leftrightarrow & \bigcap \downarrow(\lfloor (a_i, b_i) \rfloor) \subseteq \bigcap \downarrow(\lfloor (a_j, b_j) \rfloor) \\
\Leftrightarrow & \downarrow(\lfloor \{(a_i, b_i)\} \rfloor) \subseteq \downarrow(\lfloor \{(a_j, b_j)\} \rfloor) \\
\Leftrightarrow & \lfloor \{(a_i, b_i)\} \rfloor \sqsubseteq \lfloor \{(a_j, b_j)\} \rfloor \\
\Leftrightarrow & \forall j. \exists i. a_j \sqsubseteq a_i \ \&\ b_i \sqsubseteq b_j \\
\Leftrightarrow & \forall j. \exists i. [\![\varphi_j]\!] \subseteq [\![\varphi_i]\!] \ \&\ [\![\psi_i]\!] \subseteq [\![\psi_j]\!] \quad \text{by definition of } a_i, a_j, b_i \text{ and } b_j \\
\Leftrightarrow & \forall j. \exists i. \varphi_j \leq \varphi_i \ \&\ \psi_i \leq \psi_j \quad \text{by the induction hypothesis} \\
\Leftrightarrow & \forall j. \exists i. \varphi_i \to \psi_i \leq \varphi_j \to \psi_j \\
\Leftrightarrow & \bigwedge(\varphi_i \to \psi_i) \leq \bigwedge(\varphi_j \to \psi_j)
\end{array}$$

We leave it to the reader to check the cases where the formulae $\mathtt{t}_{\sigma \to \tau}$ and $\mathtt{f}_{\sigma \to \tau}$ form part of the conjunction. ■

It should be noted that the formula guaranteed to exist by the Definability Theorem is not unique. In general there are many formulae that denote the same ideal. However, the Completeness Theorem tells us that these formulae are all logically equivalent. Hence, by quotienting the set of formulae by logical equivalence we get a structure which is in bijective correspondence with the ideals of the domain $D(\sigma)$. This quotient structure is also known as the *Lindenbaum algebra* $\mathcal{LA}(\sigma)$ of the logical structure $\mathcal{L}(\sigma)$:

$$(\mathcal{LA}(\sigma), \leq_{\mathcal{LA}(\sigma)}) = (\mathcal{L}(\sigma)/ =_{\mathcal{L}(\sigma)}, \leq_{\mathcal{L}(\sigma)} / =_{\mathcal{L}(\sigma)}).$$

The Soundness and Completeness Theorems tells us that this bijection respects the order on the lattices so it is in fact a order–preserving bijection. By the results in the previous section we can conclude that for every type σ:

$$D(\sigma) \cong \mathsf{Fil}(\mathsf{Idl}(D(\sigma))) \cong \mathsf{Fil}(\mathcal{LA}(\sigma)).$$

This is the precise description of the relation between the strictness domains and the formal system. It expresses the fact that there is a bijective correspondence between elements of the domain and sets of formulae closed under conjunction and entailment.

4 Strictness analysis

This section applies the developments in the previous sections to strictness analysis. We briefly review strictness analysis by abstract interpretation and then go on to define a program logic for strictness analysis which is then proved sound and complete with respect to the abstract interpretation. For a general introduction to abstract interpretation see *e.g.*, [2].

We shall be using the simply typed lambda calculus with constants including a conditional and a fixpoint operator. The set of terms, Λ_T, of the language is given by the grammar:

$$e = x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{fix}(e) \mid \mathsf{cond}(e_1, e_2, e_3)$$

The typing rules for the language are entirely standard and omitted. The standard denotational semantics, denoted by $[\![\_]\!]^{std}$, interprets each type as a complete partial order (cpo) and each term as an element of the partial order corresponding to the type of the term. Specifically, terms of function type are interpreted as functions between cpo's.

A function f is said to be *strict* iff its result is undefined whenever its argument is undefined. This can be expressed in the standard semantics by saying that $[\![f]\!]^{std}(\bot) = \bot$, where $\bot$ stands for the smallest element in the appropriate domains. The term strictness analysis is used to denote the class of program analyses aimed at detecting when a given function is strict. We shall not go into the details of how strictness information can be used to optimise functional programs, but only refer to [12] and [3].

4.1 Strictness analysis by abstract interpretation

Strictness analysis by abstract interpretation works by interpreting the terms of Λ_T over non–standard domains built up from the two–point domain $\mathbf{2}$ interpreting the type constructor $\to$ as the domain of continuous functions. The abstract interpretation of a term f of type $Int \to Int$ would then be a function $[\![f]\!] : \mathbf{2} \to \mathbf{2}$. The function $[\![\_]\!]$ takes as arguments a term in Λ_T and an environment mapping variables to values and returns as result the abstract value of the term. It is defined as follows:

$$\begin{array}{lcl}
[\![x]\!]\rho & = & \rho(x) \\
[\![c]\!]\rho & = & \lambda x_1 \ldots . \lambda x_n . \sqcap\{x_1, \ldots, x_n\} \quad \text{where } n \geq 0 \text{ is the arity of } c \\
[\![\lambda x.e]\!]\rho & = & \lambda v.[\![e]\!]\rho[x \mapsto v] \\
[\![e_1 e_2]\!]\rho & = & [\![e_1]\!]\rho([\![e_2]\!]\rho) \\
[\![\mathsf{fix}(e)]\!]\rho & = & \sqcup_{n\in\omega} ([\![e]\!]\rho)^n(\bot) \\
[\![\mathsf{cond}(e_1, e_2, e_3)]\!]\rho & = & \begin{cases} \bot & \text{if } [\![e_1]\!]\rho = \bot \\ [\![e_2]\!]\rho \sqcup [\![e_3]\!]\rho & \text{otherwise} \end{cases}
\end{array}$$

The main result in [4] is that if the abstract interpretation of a term denotes a strict function then the standard interpretation of the function is strict as well, *i.e.*, by studying the abstract version of the function we can get information about the strictness properties of the function itself. As all domains are finite, this gives an effective method for detecting strictness properties of functions definable in the typed lambda calculus.

4.2 Strictness logic

We now define a program logic for the simply typed λ-calculus based on the formal system of strictness properties. It is shown that the program logic is equivalent to the abstract interpretation presented above in the sense that all strictness properties that holds of an abstract interpretation of a function can be proved in the strictness logic. The strictness logic is defined by the following rules

$$\textbf{Conj}\;\; \frac{\Gamma \vdash E : \psi_1 \;\; \Gamma \vdash E : \psi_2}{\Gamma \vdash E : \psi_1 \wedge \psi_2} \qquad \textbf{Weak}\;\; \frac{\Gamma \leq \Delta \;\; \Delta \vdash E : \phi \;\; \phi \leq \psi}{\Gamma \vdash E : \psi}$$

$$\textbf{Taut}\;\; \Gamma \vdash E : \mathtt{t} \qquad \textbf{Var}\;\; \Gamma[x \mapsto \phi] \vdash x : \phi$$

$$\textbf{Abs}\;\; \frac{\Gamma[x \mapsto \phi] \vdash E : \psi}{\Gamma \vdash \lambda x.E : (\phi \to \psi)} \qquad \textbf{App}\;\; \frac{\Gamma \vdash E_1 : (\phi \to \psi) \quad \Gamma \vdash E_2 : \phi}{\Gamma \vdash E_1 E_2 : \psi}$$

$$\textbf{Fix}\;\; \frac{\Gamma \vdash E : \phi \to \phi}{\Gamma \vdash \mathsf{fix}(E) : \phi}$$

$$\textbf{Cond-1}\;\; \frac{\Gamma \vdash E_1 : \mathtt{f}}{\Gamma \vdash \mathsf{cond}(E_1, E_2, E_3) : \mathtt{f}} \qquad \textbf{Cond-2}\;\; \frac{\Gamma \vdash E_2 : \phi \quad \Gamma \vdash E_3 : \phi}{\Gamma \vdash \mathsf{cond}(E_1, E_2, E_3) : \phi}$$

Other constants (including built–in functions) can be treated by adding more rules. A function like addition which is strict in both arguments is defined by the rule

$$\vdash + : \mathtt{f} \to \mathtt{t} \to \mathtt{f} \wedge \mathtt{t} \to \mathtt{f} \to \mathtt{f}.$$

The following proof demonstrates the use of the proof system. The function we analyse is taken from [11], where it was used to demonstrate the limitations of a type system without conjunctive types. It is defined as:

$$f\ x\ y\ z = \mathit{if}\ (z = 0)\ \mathit{then}\ x + y\ \mathit{else}\ f\ y\ x\ (z - 1)$$

Written in our language it looks like $\mathsf{fix}(\lambda f.\lambda x.\lambda y.\lambda z.\mathsf{cond}(z = 0, x + y, f\ y\ x\ (z - 1)))$. We want to show that this function is strict in x and y separately, which in our logic is expressed by the formula

$$\mathtt{f} \to \mathtt{t} \to \mathtt{t} \to \mathtt{f} \wedge \mathtt{t} \to \mathtt{f} \to \mathtt{t} \to \mathtt{f}$$

For notational convenience denote this formula by ψ and let similarly ψ_1 and ψ_2 denote $\mathtt{f} \to \mathtt{t} \to \mathtt{t} \to \mathtt{f}$ and $\mathtt{t} \to \mathtt{f} \to \mathtt{t} \to \mathtt{f}$, respectively. Furthermore, let E denote the term

$$\lambda x.\lambda y.\lambda z.\mathsf{cond}(z = 0, x + y, f\ y\ x\ (z - 1)).$$

and let Γ be the environment $[f : \psi, x : \mathtt{t}, y : \mathtt{f}, z : \mathtt{t}]$.
We then get the proof tree:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ {[f : \psi] \vdash E : \psi_1}\end{array} \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash x : \mathtt{t} \;\; \Gamma \vdash y : \mathtt{f}}{\Gamma \vdash x + y : \mathtt{f}} \quad \dfrac{\dfrac{\Gamma \vdash f : \psi}{\Gamma \vdash f : \psi_1} \;\; \Gamma \vdash y : \mathtt{f} \;\; \Gamma \vdash x : \mathtt{t} \;\; \Gamma \vdash (z - 1) : \mathtt{t}}{\Gamma \vdash f\ y\ x\ (z - 1) : \mathtt{f}}}{\Gamma \vdash \mathsf{cond}(z = 0, x + y, f\ y\ x\ (z - 1)) : \mathtt{f}}\,\textbf{Cond}-\textbf{2}}{[f : \psi] \vdash E : \psi_2}\,\textbf{Abs}}{[f : \psi] \vdash \lambda x.\lambda y.\lambda z.\mathsf{cond}(z = 0, x + y, f\ y\ x\ (z - 1)) : \psi}\,\textbf{Conj}}{\vdash \lambda f.\lambda x.\lambda y.\lambda z.\mathsf{cond}(z = 0, x + y, f\ y\ x\ (z - 1)) : \psi \to \psi}\,\textbf{Abs}}{\vdash \mathsf{fix}(\lambda f.\lambda x.\lambda y.\lambda z.\mathsf{cond}(z = 0, x + y, f\ y\ x\ (z - 1))) : \psi}\,\textbf{Fix}$$

4.3 Soundness and Completeness

Before stating the main theorem that relates the program logic to the abstract interpretation we introduce some notation:

Notation Let σ be a type, e a term of type σ and $\mathrm{FV}(e)$ the set of free variables in e. Assume that $d \in D(\sigma)$,$\psi \in \mathcal{LA}(\sigma)$ and let ρ and Γ be environments mapping variables to values and properties respectively.
We write:

$$\begin{array}{lcl} d \models \varphi & \equiv & d \in [\![\varphi]\!] \\ \rho \models \Gamma & \equiv & \forall x : \rho(x) \models \Gamma(x) \\ \Gamma \models e : \varphi & \equiv & \forall \rho \models \Gamma : [\![e]\!]\rho \models \varphi \end{array}$$

i.e., we overload the symbol $\models$ to mean *i*) a value satisfies a formula, *ii*) an environment of values satisfies an environment of formulae and *iii*) the denotation of a term e satisfies formula φ when e is evaluated in an environment satisfying environment Γ. The main theorem linking the abstract interpretation with the strictness logic can then be stated as follows:

Theorem 4 (Soundness & Completeness of the program logic) *Let e be a term of type σ and assume that Γ is an environment mapping free variables to propositions. For φ a formulae belonging to $\mathcal{LA}(\sigma)$ we have:*

$$\Gamma \vdash e : \varphi \Leftrightarrow \Gamma \models e : \varphi$$

Proof The soundness (the "$\Rightarrow$" in the biimplication) is proved by induction on the length of the proof. The completeness (the "$\Leftarrow$") of the logic is proved by structural induction on the term e. We treat a few of the rules here:

Soundness. For the soundness it suffices to prove that if the premises of a given rule are valid in the semantics then the conclusion is valid too. Induction then gives us that all provable facts are indeed semantically valid.

Abstraction. $\Gamma[x : \varphi] \models e : \psi$

$$\begin{array}{ll} \Rightarrow & \forall \rho \models \Gamma . \forall d \models \varphi . [\![e]\!]\rho[x \mapsto d] \in [\![\psi]\!] \\ \Rightarrow & \forall \rho \models \Gamma . \forall d \in [\![\varphi]\!] . [\![\lambda x.e]\!]\rho(d) = [\![e]\!]\rho[x \mapsto d] \in [\![\psi]\!] \\ \Rightarrow & \Gamma \models \lambda x.e : \varphi \rightarrow \psi \end{array}$$

Fixpoints. $\Gamma \models e : \psi \rightarrow \psi$

$$\begin{array}{ll} \Rightarrow & \forall \rho \models \Gamma . \forall i \in \omega . [\![e]\!]^i \rho(\bot) \in [\![\psi]\!] \quad \text{since } \bot \in [\![\psi]\!] \\ \Rightarrow & \forall \rho \models \Gamma . [\![\mathsf{fix}(e)]\!]\rho \in [\![\psi]\!] \quad \text{since } [\![\psi]\!] \text{ is closed under } \sqcup \\ \Rightarrow & \Gamma \models \mathsf{fix}(e) : \psi \end{array}$$

Cond-1. $\Gamma \models e_1 : \mathtt{f}$

$$\Rightarrow \forall\rho \models \Gamma.[\![e_1]\!]\rho = \bot$$
$$\Rightarrow \forall\rho \models \Gamma.[\![\mathsf{cond}(e_1, e_2, e_3)]\!]\rho = \bot$$
$$\Rightarrow \Gamma \models \mathsf{cond}(e_1, e_2, e_3) : \mathtt{f}$$

Completeness Completeness amounts to showing (by structural induction over the terms) that all facts that hold in the semantics can be proved in the logic.

Application. $\Gamma \models e_1 e_2 : \psi$

$$\Rightarrow \forall\rho \models \Gamma.[\![e_1 e_2]\!]\rho = [\![e_1]\!]\rho([\![e_1]\!]\rho) \in [\![\psi]\!]$$
$$\Rightarrow \forall\rho \models \Gamma.d \in \downarrow([\![e_2]\!]\rho) \Rightarrow [\![e_1]\!]\rho(d) \in [\![\psi]\!]$$
$$\Rightarrow \forall\rho \models \Gamma.\exists\varphi.[\![e_2]\!]\rho \in [\![\varphi]\!] \;\&\; [\![e_1]\!]\rho \in [\![\varphi \to \psi]\!] \quad \text{(Definability Theorem)}$$
$$\Rightarrow \Gamma \vdash e_1 : \varphi \to \psi \;\&\; \Gamma \vdash e_2 : \varphi \quad \text{by induction hypothesis}$$
$$\Rightarrow \Gamma \vdash e_1 e_2 : \psi$$

Abstraction. $\Gamma \models \lambda x.e : \varphi \to \psi$

$$\Rightarrow \forall\rho \models \Gamma.[\![\lambda x.e]\!]\rho \in [\![\varphi \to \psi]\!]$$
$$\Rightarrow \forall\rho \models \Gamma.\exists d_1, d_2.[\![\varphi]\!] = \downarrow(d_1) \;\&\; [\![\psi]\!] = \downarrow(d_2) \;\&\; [\![e]\!]\rho[x \mapsto d_1] \sqsubseteq d_2$$
$$\Rightarrow \Gamma[x : \varphi] \models e : \psi$$
$$\Rightarrow \Gamma \vdash \lambda x.e : \varphi \to \psi$$

Fixpoints. $\Gamma \models \mathsf{fix}(e) : \psi$

$$\Rightarrow \forall\rho \models \Gamma.\downarrow([\![\mathsf{fix}(e)]\!]\rho) \subseteq [\![\psi]\!]$$

Now, let φ be such that $[\![\varphi]\!] = \downarrow([\![\mathsf{fix}(e)]\!]\rho)$. As $[\![\varphi]\!] \subseteq [\![\psi]\!]$ we have from the completeness theorem for the domain logic that $\varphi \leq \psi$, so we can deduce as follows:

$$[\![e]\!]\rho([\![\mathsf{fix}(e)]\!]\rho) = [\![\mathsf{fix}(e)]\!]\rho$$
$$\Rightarrow \Gamma \vdash e : \varphi \to \varphi \quad \text{by induction hypothesis}$$
$$\Rightarrow \Gamma \vdash \mathsf{fix}(e) : \varphi \;\&\; \varphi \leq \psi$$
$$\Rightarrow \Gamma \vdash \mathsf{fix}(e) : \psi \quad \text{by the rule } \textbf{Weak} \qquad \blacksquare$$

By combining this theorem with the isomorphism between semantic domains and filters of formulae $\eta_\sigma : D(\sigma) \to \mathsf{Fil}(\mathcal{LA}(\sigma))$ we can give a precise characterisation of the correspondence between the abstract interpretation and the strictness logic. We have that for an expression e of type σ and for environments $\rho \models \Gamma$

$$[\![e]\!]\rho = \eta^{-1}\{\varphi \mid \Gamma \vdash e : \varphi\}$$

i.e., that the denotation of an expression given by the abstract interpretation is determined and determines the set of formulae provable of e in the strictness logic.

5 Conclusion

We have presented a framework for relating abstract interpretation and non–standard type systems. The presentation took the abstract interpretation as starting point and developed a program logic equivalent to the interpretation. It is important to notice that this process is reversible, because this is the basis for demonstrating that the two methods are equally powerful. We believe that the representation of lattices by certain classes of subsets (here ideals) is the basis for relating denotationally and axiomatically formulated program analyses. It should be noted that the representation of join–semilattices by their ideals presented here is a particular instance of a general mathematical construction known as *Stone duality*. In fact the proposition linking the strictness domains $D(\sigma)$ with the formal system $\mathcal{L}(\sigma)$ is a first step towards establishing a natural transformation forming an adjunction between the two functors Fil and Idl. It is well outside the scope of this paper to go into more details about this subject and we shall only refer the reader to [1], [5] and [9].

The work reported here is of course inspired by previous work on strictness analysis via type inference reported by Kuo and Mishra in [11]. The difference between our work and theirs is that whereas they concentrate on obtaining a good implementation of their type system, we have addressed the question of how these type systems arise and why they give correct results. Thus we see our work as complementary to the work in [11]. It is evident that the developments reported here are of limited use if they are not accompanied by advances in the technology of implementing these program logics. Fortunately, work is going on in this area, see *e.g.*, [6].

We have not made any attempt to prove the strictness logic correct with respect to a standard semantics for the language. This is because once we have established the equivalence to the abstract interpretation we can rely on previous correctness results for the abstract interpretation. Still, abstract interpretation uses abstraction maps from the standard to the abstract domains to establish the correctness of the analysis, and an obvious question is what is the corresponding concept on the logical side? It is a fact that we can represent a Scott–domain by the so–called irreducible, closed subsets. We conjecture that it should be possible from an axiomatic semantics based on these irreducible, closed subsets to view abstraction as equating certain closed sets in the standard domain. On the logical side this means that abstraction simply amounts to introducing logical equivalences between properties in the standard semantics, but this needs further investigation.

Finally it would be a natural continuation of the work reported here to apply similar ideas to other program analyses. We are currently investigating the relationship between the binding–time analysis via type–inference described in [10] and the PER–based abstract interpretation presented in [8]. Another analysis worth considering is the system of use–types suggested in [14] which is aimed at estimating how many times a given value is used during execution of a program. There is currently no corresponding abstract interpretation for this kind of analysis so it would be an interesting exercise to see whether the principles presented in this paper can be used to design such an abstract interpretation.

Acknowledgements

Thanks are first and foremost due to Sebastian Hunt for his continued interest in the work reported here. His insight and his helpful comments have been a great support during the whole work process. Thanks are also due to Geoff Burn, Chris Hankin, Dave Sands and the partners in the SEMANTIQUE project for listening to my ideas and suggesting improvements.

References

[1] S. Abramsky. Domain theory in logical form. *Journal of Pure and Applied Logic*, 1990.

[2] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[3] G.L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages.* PhD thesis, Imperial College, University of London, 1987.

[4] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.

[5] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[6] C. K. Gomard. Partial type inference for untyped functional programs. In *Lisp and Functional Programming '90*, pages 282–287. ACM, 1990.

[7] S. Hunt and C. Hankin. Fixed points and frontiers: A new perspective. *Journal of Functional Programming*, 1(1), 1991.

[8] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics–Based Program Manipulation*, 1991.

[9] P. T. Johnstone. *Stone Spaces*, volume 3 of *Cambridge Studies in Advanced Mathematics.* Cambridge University Press, Cambridge, 1982.

[10] N. Jones, C. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *International Conference on Computer Languages.* IEEE Computer Society, 1990.

[11] T-M Kuo and P. Mishra. Strictness analysis : A new perspective based on type inference. In *Proc. 4th. Int. Conf. on Functional Programming and Computer Architecture.* ACM Press, 1989.

[12] A. Mycroft. *Abstract Interpretation and Optimising Transformation for Applicative Programs.* PhD thesis, Univ. of Edinburgh, 1981.

[13] D. A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[14] P. Wadler. Is there a use for linear logic? In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics–Based Program Manipulation*, 1991.