Type and Effect Systems via Abstract Interpretation*

Jérôme Vouillon
Pierre Jouvelot

CRI, Ecole des Mines de Paris
{vouillon, jouvelot}@cri.ensmp.fr

July 12, 1995

Abstract

Abstract interpretation and type and effect systems are two well-known frameworks for the static analysis of programs. While abstract interpretation uses the program control-flow to approximate its run-time behavior, type and effect systems are based on the program structural syntax.

The fundamental result of this paper is to show how these *a priori* distinct approaches can be related to each other, shedding a new light on their relative expressiveness. A single example of static analysis, namely a straightforward program time complexity estimator for the simply typed lambda-calculus with recursion, is used throughout the paper to present the core ideas. We show how an abstract interpretation and a type and effect system can be designed for this analysis and prove them equivalent, thus paving the way to a better understanding of the relative merits of these two frameworks.

Keywords: Static analysis, abstract interpretation, type and effect systems, foundations of program semantics, time complexity.

1 Introduction

Abstract interpretation [2] and type and effect systems [6] are two well-known frameworks for the static analysis of programs. Although targeting the same goal, *i.e.*, obtaining an as-good-as possible understanding of the program run-time behavior at compile time, these two frameworks are based on quite different grounds:

- Abstract interpretation uses the program control-flow to analyze its run-time behavior. Identifiers are bound to elements of a lattice of abstract values, and (generally uncomputable) fixpoint computations are approximated to ensure termination of the analysis.
- Type and effect systems are based on the program structural syntax. Each expression of the program is associated to a type and an effect, respectively abstracting the value it evaluates to, and the side-effects it performs during evaluation. Effects, which are also included within function types, are usually defined over an algebraic domain [5, 11]. This requires non-free unification to be performed during type and effect reconstruction.

Even though these two approaches seem quite different, the fundamental result of this paper is to show that these frameworks can indeed be formally related to each other, shedding a new light on their relative expressiveness.

*This paper is partially supported by the ESPRIT BRA project LOMAPS (8130).

To motivate and simplify the presentation, a single example of static analysis, namely a straightforward program time complexity estimator for the simply typed lambda-calculus with recursion [4, 14], is used throughout the paper to present the core ideas. In this system, an approximation of the number of function applications and variable lookups is obtained for each non-recursive expression, the special value $\top$ being used for recursive ones[1]. Note that this assignment is by no means trivial since the full higher-order language is treated and expressions that use the fixpoint constructor μ but are not actually recursive are properly treated.

Both a type and effect system and an abstract interpretation for this static analysis are given. Our main theorem, Theorem 3, shows how these two specifications are related, effectively allowing one to deduce the type and effect system from the abstract interpretation or vice-versa. This is the first time such a formal and clear connection is established between these two domains.

The structure of the paper is the following. In Section 2, we give the general notations used in the paper. Section 3 defines the object language and gives a dynamic semantics instrumented with time information. Section 4 recalls the static type and effect system semantics for program complexity analysis defined in [4]. Section 5, after a brief introduction on Galois connections, is dedicated to the definition of a new abstract interpretation for complexity analysis: (1) abstraction function for values and (2) abstract semantics. Section 6 formally relates these two static semantics, introducing the main Theorem 3. Section 7 suggests possible extensions of this work with respect to subtyping and polymorphism. Before concluding in Section 9, we survey the related work in Section 8.

All proofs are omitted from this abstract, but can be obtained by anonymous ftp on `cri.ensmp.fr` *in* `pub/popl96-submission.dvi`.

2 Definitions and Notations

We here give general definitions and explain notations used throughout the paper. We note $A \rightharpoonup B$ the set of total functions from the set A into the set B, while $A \nrightarrow B$ is extended to *partial* ones.

Let X and Y be two partially ordered sets. We note $\sup(x, y)$ the least upper bound, if it exists, of x and y. Let ϕ be a function from X to Y, noted $\phi : X \rightharpoonup Y$:

- ϕ is *monotone*, which is written $\phi : X \xrightarrow{m} Y$, iff $\forall u, v \in X, u \leq v \Rightarrow \phi(u) \leq \phi(v)$.
- ϕ is *additive*, which is written $\phi : X \xrightarrow{a} Y$, iff for any sequence $(u_i)_{i \in I}$ of elements u_i of X such that $\sup_{i \in I} u_i$ exists, $\sup_{i \in I} \phi(u_i) = \phi(\sup_{i \in I} u_i)$.

One can notice that an additive function is monotone.

We note $f[x \leftarrow v]$ the function that returns v for x and $f\ y$ for y different from x. We sometimes note $f : x \longmapsto v$ when $f\ x = v$.

3 Language Definition

3.1 Syntax

We use expressions (in Exp) of the lambda calculus with recursion as our language of study. For the sake of simplicity in the dynamic semantics, we only allow recursion to be used on explicit function definitions. The classical language syntax is given below:

[1] [4] used the term `long` for $\top$.

$$
\begin{array}{lll}
e \in Exp ::= & x & \text{Identifier in } I \\
& \lambda x.e & \text{Function definition} \\
& \mu f.\lambda x.e & \text{Recursive function} \\
& e\,e & \text{Function application}
\end{array}
$$

3.2 Dynamic Semantics

Since we use time complexity as our example in this paper, the dynamic semantics of this language departs from the usual one in its time instrumentation. Every expression, in an environment $\rho : I \nrightarrow V$ from identifiers to values, evaluates to a pair formed from a value and an integer in $\mathbb{N}$ denoting the number of evaluation steps performed during evaluation. Without loss of generality, we assume that all operations in the dynamic semantics take an amount of time equal to 1. The definition of values in V is as follows:

$$
\begin{array}{lll}
v \in V ::= & \mathbf{b} & \text{Basic value} \\
& \langle \lambda x.e \rangle_\rho & \text{Closure}
\end{array}
$$

For our purpose, we do not need to distinguish between different basic values; they are all gathered under the name $\mathbf{b}$.

The dynamic semantics $[\![.]\!] : Exp \to (I \nrightarrow V) \nrightarrow (I \times \mathbb{N})$ specifies how an expression e is evaluated in an environment ρ to yield a value v and time n. We use eager evaluation since it is easier to define a dynamic semantics with time within this strategy. It is inductively defined as follows:

$$
\begin{array}{l}
[\![x]\!]\rho = (\rho(x), 1) \\
[\![\lambda x.e]\!]\rho = (\langle \lambda x.e \rangle_\rho, 1) \\
[\![e_1 e_2]\!]\rho = (v'', 1 + n + n' + n'') \text{ if } \left\{ \begin{array}{l} [\![e_1]\!]\rho = (\langle \lambda x.e \rangle_{\rho'}, n) \\ [\![e_2]\!]\rho = (v', n') \\ [\![e]\!]\rho'[x \leftarrow v'] = (v'', n'') \end{array} \right. \\
[\![\mu f.\lambda x.e]\!]\rho = (v, 1) \text{ where } v = \langle \lambda x.e \rangle_{\rho[f \leftarrow v]}
\end{array}
$$

The two following sections define two different static semantics that strive to approximate this dynamic semantics. The first one (Section 4) uses the framework of effect systems, while the second (Section 5) is based on abstract interpretation. The main result of Section 6 formally shows how they relate.

4 Effect System

This section quickly recalls the material presented in [4]. Since we are not specifically interested here in typing issues, we restrict ourselves to expressions that can be typed using the rules of the simply-typed lambda-calculus [8].

This static system extends the classical simply-typed lambda calculus with effect information. The type $\mathbf{B}$ corresponds to basic values $\mathbf{b}$. Function types are decorated with *latent* effect information; this encapsulates the effect of the function body. In our example, effects, *i.e.*, times, are elements of the non-free algebra $\mathbb{N}^\top$ that extends $\mathbb{N}$ with the value $\top$. Possibly non-terminating expressions have time $\top$, while certainly terminating ones correspond to usual integers. Usual arithmetic operations are extended to $\mathbb{N}^\top$, $\top$ being an absorbing element (e.g., $\top + 1 = \top$). The domains of types and effects are defined below:

$$
\begin{array}{lll}
t \in T ::= & \mathbf{B} & \text{Basic type} \\
& t \xrightarrow{\epsilon} t & \text{Function type}
\end{array}
$$

$$
\begin{array}{lll}
\epsilon \in \mathbb{N}^\top ::= & n & \text{Integer} \\
& \top &
\end{array}
$$

A type and time system for our language specifies how every expression e in a typing environment $A : I \rightharpoonup T$ from identifiers to types in T can be associated to a type t and effect ϵ in $\mathbb{N}^{\top}$. The specification of the semantics, using operational rules in the style of [13], is given in Figure 1.

$$\frac{A\ x = t}{A \ \vdash \ x \ : \ t, 1}$$

$$\frac{A[x \leftarrow t'] \ \vdash \ e \ : \ t, \epsilon}{A \ \vdash \ \lambda x.e \ : \ t' \xrightarrow{\epsilon} t, 1}$$

$$\frac{A[f \leftarrow t \xrightarrow{\epsilon} t'] \ \vdash \ \lambda x.e \ : \ t \xrightarrow{\epsilon} t', 1}{A \ \vdash \ \mu f.\lambda x.e \ : \ t \xrightarrow{\epsilon} t', 1}$$

$$\frac{\begin{array}{c} A \ \vdash \ e_1 \ : \ t_2 \xrightarrow{\epsilon} t, \epsilon_1 \\ A \ \vdash \ e_2 \ : \ t_2, \epsilon_2 \end{array}}{A \ \vdash \ e_1 e_2 \ : \ t, \epsilon + \epsilon_1 + \epsilon_2 + 1}$$

Figure 1: Static Semantics

The interplay between types and effects can be easily seen by looking at the rules for abstraction and application. In the abstraction case, the effect of the lambda body is encapsulated within the function type; this effect is then exposed when this function is applied, as shown by the rule for function application. The case for recursive definition is a simple extension of the abstraction rule.

5 Abstract Interpretation

We now use the formalism of abstract interpretation to specify an analysis that, as we latter show, computes information equivalent to the one of Section 4. But first, we recall some basic definitions and properties of the formal foundations of abstract interpretation.

5.1 Galois Connections

Let $(E, \leq)$ and $(E^\sharp, \leq^\sharp)$ be two partial orders. When the order relation is clear from context, we simply write E instead of $(E, \leq)$.

Definition 1 *Let $\alpha : E \rightarrow E^\sharp$ and $\gamma : E^\sharp \rightarrow E$. (α, γ) is a* Galois connection, *which is written $E \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} E^\sharp$, iff*

$$\forall p \in E, \forall p^\sharp \in E^\sharp, \alpha(p) \leq^\sharp p^\sharp \Leftrightarrow p \leq \gamma(p^\sharp)$$

The *abstraction* function α maps values from the concrete domain E into the abstract one $E^\sharp$, while the *concretization* function γ goes in the opposite direction. This definition expresses the conservative nature of the abstraction process; this correctness criterion can be better seen in the following corollary:

$$p \leq (\gamma \circ \alpha)(p) \text{ and } (\alpha \circ \gamma)(p^\sharp) \leq p^\sharp.$$

One can easily show that α and γ are additive (and thus monotone), and that the following duality relation holds:

If $(E, \leq) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (E', \leq')$, then $(E', \geq') \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} (E, \geq)$

Let us look at a few examples of Galois connections, which are used later on:

- $(\mathcal{P}(E), \subseteq) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (\mathcal{P}(E), \supseteq)$ shows how one can define an abstraction over sets of values: the abstraction of a set S is the set of the upper bounds of all elements in S:

$$\begin{cases} \alpha(S) & = & \{n' \in E / \forall n \in S, n \leq n'\} \\ \gamma(S') & = & \{n \in E / \forall n' \in S', n \leq n'\} \end{cases}$$

- $\mathcal{P}(I \rightarrow E) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} I \rightarrow \mathcal{P}(E)$ maps sets of functions (using set inclusion as partial order) from I to E into functions from I to sets of elements of E (using set inclusion on the function range as partial order):

$$\begin{cases} \alpha(F) & = & \lambda x.\{f(x) / f \in F) \\ \gamma(f^\sharp) & = & \{f : I \rightarrow E / \forall x \in I, f(x) \in f^\sharp(x)\} \end{cases}$$

- The last example $(\mathcal{P}(A \times B), \subseteq) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \mathcal{P}(A) \times \mathcal{P}(B)$ extends Galois connections to products of sets:

$$\begin{cases} \alpha(R) & = & (\{a \in A / \exists b \in B, (a, b) \in R\}, \{b \in B / \exists a \in A, (a, b) \in R\}) \\ \gamma(X, Y) & = & X \times Y \end{cases}$$

New connections can also be designed by using an existing Galois connection, say $E \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} E'$, and straightforwardly extending it to the sets $I \rightarrow E$ and $I \rightarrow E'$, or $E \times F$ and $E' \times F$. Galois connections can also be composed in order to obtain some more complex Galois connections:

$E \underset{\gamma \circ \gamma'}{\overset{\alpha' \circ \alpha}{\rightleftarrows}} E''$ if $E \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} E' \underset{\gamma'}{\overset{\alpha'}{\rightleftarrows}} E''$

5.2 Value Abstraction

We first show how to abstract values of V. To make the definition more symmetric with respect to abstraction and concretization, we reason in terms of sets of values, instead of simple values. It is thus natural to define the abstraction that way:

$$\alpha(\nu) = \bigcap_{v \in \nu} \alpha_V(v)$$

where $\alpha_V : V \rightarrow \mathcal{P}(T)$ is still to be specified. We deduce from that (see [2]):

$$\gamma(\tau) = \{v \in V / \tau \subseteq \alpha_V(v)\}$$

The central idea is then to see the abstraction of a value v as the set of ground types compatible (in a sense to be made more precise later on) with it; $(\mathcal{P}(V), \subseteq) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (\mathcal{P}(T), \supseteq)$ is then a Galois connection. Note that one needs to use $\supseteq$ as the partial order on $\mathcal{P}(T)$ since, informally, as a set a values grows, its set of compatible types shrinks.

Let us now define α_V by carefully comparing the definitions of the dynamic and static semantics. First, the concrete semantics given in Section 3 defines a notion of application on values:

$$\begin{array}{rcccccl} \phi & : & V & \to & V & \not\to & V \times \mathbb{N} \\ & & v & \longmapsto & v' & \longmapsto & (v'', n'') \\ & & & & & & \text{if } v = \langle \lambda x.e \rangle_\rho \\ & & & & & & \text{and } [\![e]\!]\rho'[x \leftarrow v'] = (v'', n'') \end{array}$$

Indeed, we have:

$$[\![e_1 e_2]\!]\rho = (v'', 1 + n + n' + n'')$$
$$\text{if } \begin{cases} (v, n) = [\![e_1]\!]\rho \\ (v', n') = [\![e_2]\!]\rho \\ (v'', n'') = \phi(v)(v') \end{cases}$$

Second, there is also a quite natural notion of application on sets of types induced by the static semantics of Section 4:

$$\begin{array}{rcccccl} \psi & : & \mathcal{P}(T) & \xrightarrow{a} & \mathcal{P}(T) & \xrightarrow{a} & \mathcal{P}(T \times \mathbb{N}^\top) \\ & & \tau & \longmapsto & \tau' & \longmapsto & \{(t, \epsilon) / \exists t' \in \tau', t' \xrightarrow{\epsilon} t \in \tau\} \end{array}$$

So to define an abstraction interpretation that mimics the static semantics, we need to choose α_V such that ψ is an abstraction of ϕ. For that purpose, it is necessary to insure the following commuting safety condition:

$$(1) \qquad \forall v \in V, \forall v' \in V, \phi(v)(v') \in \gamma_R(\psi(\alpha(\{v\}))(\alpha(\{v'\})))$$

where γ_R (resp. α_R) is the extension of γ (resp. α) to sets of pairs of types (resp. values) and times computed by the static (resp. dynamic) semantics. In words, the concretization of the return types of the applications of function types in the abstraction of v to the argument types in the abstraction of v' contains (*i.e.*, is a conservative approximation of) the value computed by applying the function v to the argument v'. A similar line of explanation can be thought out for time information.

Definition 2 ***(Abstraction Function).***

$$\begin{array}{rcl} \alpha_V(\langle \lambda x.e \rangle_\rho) & = & \left\{ t \xrightarrow{\epsilon''} t' / \begin{array}{l} \forall \nu' \in \mathcal{P}(V), t \in \alpha(\nu') \Rightarrow \\ (t', \epsilon'') \in \alpha_R(\{[\![e]\!]\rho[x \leftarrow v'] / v' \in \nu'\}) \end{array} \right\} \\ \alpha_V(\mathbf{b}) & = & \{\mathbf{B}\} \end{array}$$

$$\alpha_R(\mathcal{R}) = \alpha(\{v \in V / \exists n \in \mathbb{N}, (v, n) \in \mathcal{R}\}) \times \{\epsilon \in \mathbb{N}^\top / \forall (v, n) \in \mathcal{R}, n \sqsubseteq \epsilon\}$$

where $\epsilon \sqsubseteq \epsilon'$ iff $\epsilon = \epsilon'$ or $\epsilon' = \top$. This relation order on effects is required to properly deal in the abstract semantics with possibly non-terminating expressions. It is interesting to notice how explicit the treatment of non-termination is in the abstract interpretation framework. In the static semantics based on effect systems, the only thing that forces looping constructs to admit a time $\top$ is the algebraic rules of the effect domain, and the equality of ϵ with $\epsilon + n$ for some n induced by recursive calls.

Note that it is not obvious that this recursive definition of α_V is well-formed; an argument based on the inductive size of types can be used to actually show that this is indeed the case.

Theorem 1 *When α_V is defined as above, ψ is an abstraction of ϕ.*

Also, it is worth mentioning that the proof itself imposes stringent constraints on the proper definition of α_V. In fact, except for base types and values, the viable definitions for α_V are always smaller than the one given above, which is thus as precise as possible.

5.3 Abstract Semantics

The abstract semantics based on abstract interpretation maps expressions in a given abstract environment to sets of pairs, each of which is made of a type and a time. So we want to have an abstract semantics such as:

$$[\![e]\!]^\sharp : (I \to \mathcal{P}(T)) \to \mathcal{P}(T \times \mathbb{N}^\top)$$

since $\mathcal{P}(T)$ is the set of abstract values. The Galois connection defined in the previous subsection induces the two following Galois connections:

- $\mathcal{P}(I \twoheadrightarrow V) \underset{\gamma_E}{\overset{\alpha_E}{\rightleftarrows}} I \to \mathcal{P}(T)$ relates abstract environments to environments in the dynamic semantics:
$$\begin{cases} \alpha_E(P)(x) = \alpha(\{\rho(x)/\rho \in P\}) \\ \gamma_E(\rho^\sharp) = \{\rho : I \twoheadrightarrow V / \forall x \in \mathrm{dom}\,\rho, \rho(x) \in \gamma(\rho^\sharp(x))\} \end{cases}$$

- $(\mathcal{P}(V \times \mathbb{N}), \subseteq) \underset{\gamma_R}{\overset{\alpha_R}{\rightleftarrows}} (\mathcal{P}(T \times \mathbb{N}^\top), \supseteq)$ connects the returned pairs in the dynamic semantics with their compatible type and time (see above the definition of α_R):
$$\gamma_R(\mathcal{R}^\sharp) = \gamma(\{t \in T / \exists \epsilon \in \mathbb{N}^\top, (t, \epsilon) \in \mathcal{R}^\sharp\}) \times \{n \in \mathbb{N} / \forall (t, \epsilon) \in \mathcal{R}^\sharp, n \sqsubseteq \epsilon\}$$

according to the following schemes:

$$\mathcal{P}(I \twoheadrightarrow V) \;\rightleftarrows\; I \to \mathcal{P}(V) \;\rightleftarrows\; I \to \mathcal{P}(T)$$

$$\mathcal{P}(V \times \mathbb{N}) \;\rightleftarrows\; \mathcal{P}(V) \times \mathcal{P}(\mathbb{N}) \;\rightleftarrows\; \mathcal{P}(T) \times \mathcal{P}(\mathbb{N}^\top) \;\rightleftarrows\; \mathcal{P}(T \times \mathbb{N}^\top)$$

We are now equipped to define our abstract interpretation in the way given in Figure 2.

$$[\![x]\!]^\sharp \rho^\sharp = \rho^\sharp(x) \times \{1\}$$
$$[\![\lambda x.e]\!]^\sharp \rho^\sharp = \{t \xrightarrow{\epsilon''} t' \in T / (t', \epsilon'') \in [\![e]\!]^\sharp \rho^\sharp[x \leftarrow \{t\}]\} \times \{1\}$$
$$[\![e_1 e_2]\!]^\sharp \rho^\sharp = \{(t', 1 + \epsilon + \epsilon' + \epsilon'') / \exists t \in T, (t \xrightarrow{\epsilon''} t', \epsilon) \in [\![e_1]\!]^\sharp \rho^\sharp \wedge (t, \epsilon') \in [\![e_2]\!]^\sharp \rho^\sharp\}$$
$$[\![\mu f.\lambda x.e]\!]^\sharp \rho^\sharp = \{(t, 1) / t \in T \wedge (t, 1) \in [\![\lambda x.e]\!]^\sharp \rho[f \leftarrow \{t\}]\}$$

Figure 2: Abstract Semantics

This abstract interpretation computes, for each expression, the set of types and times that it can have.

One now needs to ensure that the abstract values computed by this abstract interpretation is correct with respect to the dynamic semantics. Note that we did not do it for the static semantics of Section 4, since this result is already present in [4].

An abstract semantics $[\![.]\!]^\sharp$ is correct with respect to the dynamic semantics whenever:

$$\forall e \in Exp, \forall \rho \in I \twoheadrightarrow V, [\![e]\!]\rho \in \gamma_R([\![e]\!]^\sharp(\alpha_E(\{\rho\})))$$

that is whenever the abstract semantics does not forget any values with respect to the dynamic semantics. Thus,

Theorem 2 *The abstract semantics of Figure 2 above is correct.*

6 Equivalence

Since we wanted to establish a connection between a type and effect system and abstract interpretation, we are left with comparing the two analysis given in Figures 1 and 2. In fact, this relationship can be stated by the following fundamental theorem:

Theorem 3 *Let A be a typing environment and define $\rho^\sharp$ the abstract environment such that $\rho^\sharp x$ is $\{Ax\}$ if x is in the domain of A, and $\emptyset$ otherwise. Then:*

$$A \vdash e \ : \ t, \epsilon \ \textit{iff} \ (t, \epsilon) \in [\![e]\!]^\sharp \rho^\sharp$$

This theorem precisely define the relationship between the formulations of a time analysis respectively by abstract interpretation and via a type and effect system.

We end this section with a couple of observations trying to recapitulate how we analyze the difference between these two frameworks. First, abstract interpretation can be seen as a specification method in which one manipulates the *set* of possible abstract values, here types. Type and effect systems, on the other end, use non-determinism to assign a proper type and effect to a given expression. So, for instance, one could think of introducing higher-order constructs in an abstract interpretation (e.g., taking the maximum of all possible types) which could be difficult to express in a static semantics framework.

Second, the manipulation of recursion is usually much more explicit in an abstract interpretation than in a static semantics, as can be seen in Figure 1. However, this is not quite true for the abstract interpretation of Figure 2, which should be expected since both specifications are equivalent.

Last, it is interesting to relate the fundamental concepts of both frameworks. For instance, the use of name equality in the static semantics rules (e.g., in the application case) which forces two types to be equal relates to set intersection in the abstract interpretation world (i.e, a type must belong to two abstract values). Also, one might be surprised not to find the usual abstract call to function bodies while dealing with function applications in the abstract interpretation framework; in fact, the abstract semantics does not need here to mimic the usual flow of control since its result can be precomputed by our choice of abstract value for closures.

7 Extensions

Two obvious contenders for extension are polymorphism and subtyping/subeffecting. We look at these issues below.

7.1 Polymorphism

The first problem one may notice is that T contains in fact no type variables, but only a type constant $\mathbf{B}$; there is thus no hope in this setting to have the usual polymorphism in the static semantics. But as the abstract semantics manipulates sets of types, we saw in Section 6 this enables operations which are not expressible in a static semantics. So, polymorphic types can be here represented as sets of ground types (the sets of their ground instances). One naturally gets the following specification:

$$[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!]^\sharp \rho^\sharp = \{(t', 1 + \epsilon + \epsilon')/\epsilon \in \mathcal{E} \wedge (t', \epsilon') \in [\![e_2]\!]^\sharp \rho^\sharp [x \leftarrow \tau]\}$$
$$\text{where} \begin{cases} \tau & = & \{t \in T / \exists \epsilon \in \mathbb{N}^\top, (t, \epsilon) \in [\![e_1]\!]^\sharp \rho^\sharp\} \\ \mathcal{E} & = & \{\epsilon \in \mathbb{N}^\top / \exists t \in T, (t, \epsilon) [\![e_1]\!]^\sharp \rho^\sharp\} \end{cases}$$

It is not completely clear yet how this could be reflected into a given static semantics (the previous equation computes what amounts to a most general type), although one may look at an approach similar to the first-class polymorphic values of [12].

7.2 Subtyping

Effects are ordered. For the sake of simplicity, we did not introduce subeffecting (allowing an expression to admit larger times) in the static semantics, although it would be quite easy to add it, and extend the abstract interpretation accordingly.

It seems then quite natural to try to extend this order to types. The idea is that a type is greater than another whenever it can be used instead of this other, that is, in the abstraction framework:

$$\forall t, t' \in T, t \leq t' \Rightarrow (\forall \nu \in \mathcal{P}(V), t \in \alpha(\nu) \Rightarrow t' \in \alpha(\nu)) \tag{2}$$

It seems intuitive that the following contravariant order satisfies this constraint:

$$\frac{\begin{array}{rcl} t_2' & \leq & t_1' \\ t_1 & \leq & t_2 \\ \epsilon_1 & \sqsubseteq & \epsilon_2 \end{array}}{t_1' \xrightarrow{\epsilon_1} t_1 \leq t_2' \xrightarrow{\epsilon_2} t_2}$$

Let us define the closure operator $c : \mathcal{P}(T) \to \mathcal{P}(T)$ by:

$$c(\tau) = \{t' \in T / \exists t \in \tau, t \leq t'\}$$

The condition (2) can be proved equivalent to

$$c \circ \alpha = \alpha \tag{3}$$

The interesting aspect of this equation is that, whenever we have the approximation τ of $\alpha(\nu)$, we know that in fact $c(\tau)$ is also an approximation (as $c(\tau) \subseteq c(\alpha(\nu)) = \alpha(\nu)$).

This can be used to prove a less restrictive abstract semantics for application:

$$[\![e_1 e_2]\!]^\sharp \rho^\sharp = \left\{ \begin{array}{l} (t_1', 1 + \epsilon + \epsilon' + \epsilon'')/ \\ \exists t_1, t_2 \in T, (t_1 \xrightarrow{\epsilon''} t_1', \epsilon) \in [\![e_1]\!]^\sharp \rho^\sharp \wedge (t_2, \epsilon') \in [\![e_2]\!]^\sharp \rho^\sharp \wedge t_2 \leq t_1 \end{array} \right\}$$

The rewriting rule given in Section 6 can be applied here, giving the usual application with subtyping for the actual argument.

8 Related Work

One of the first papers relating type systems and abstract interpretation is [10], which uses the similar idea of representing types by sets of values, and formally describing an abstract interpretation that computes them. The equivalence was not formally proved. Our work extends this approach to add effect information and ability to deal with non-free algebras. Also, their approach uses a relational model instead of the Galois connections we introduced.

Mycroft and Jones were actually interested in strictness analysis in the previous paper. This example is also used by Coppo and Ferrari [1]. Here, they use filter domains within their abstract semantics, and are able to relate them to the intersection types used for strictness analysis. We use much simpler domains for our abstract interpretation, and study a different type semantics.

We discussed in Section 7 the possibility of adding polymorphism in both frameworks. This line of thought is present in the work of Monsuez [9], although he is working with a much more complex system. We believe the presented approach is simpler and, of course, adds the orthogonal issues of effect information.

The paper [3] discusses the interplay between abstract interpretation and inductive definitions. However the purpose there is to find a way of relating different static semantics, using abstract interpretation as a vehicle for transition. We are interested in a more applied approach, trying to better explain the relative power of existing frameworks.

Note that most of these papers relate abstract interpretation and type (only) systems. Although this is an interesting endeavor, we believe it actually makes more sense to compare it with type *and* effect systems. Indeed, the various effect systems in the literature (e.g., [11, 16]) study various behavioral abstractions of programs in a way similar to the various related analyses performed by abstract interpretation (e.g., [7, 15]).

9 Conclusion

We showed in this paper how the *a priori* distinct approaches of abstract interpretation and type and effect systems can be related to each other, shedding a new light on their relative expressiveness. We used a simple program complexity analysis as a driving line for our study. We showed how an abstract interpretation and a type and effect system can be designed for this analysis and proved them equivalent. We also suggested possible extensions of what we believe to be a simple and easy to understand way of comparing these two frameworks, in particular with respect to subtyping and polymorphism.

It is worth mentioning that, surprisingly, our work does not heavily rely on the actual effects described. The abstraction of values, for instance, only needs to use the fact that the effect domain is ordered. This makes us hope that a more general presentation is not too far away, abstracted over the actual effect analysis used in the static semantics (or abstract interpretation).

Acknowledgments

The second author thanks Patrick Cousot for discussions, at a workshop in Bordeaux (long) time ago, about some of the issues discussed in this paper.

References

[1] Coppo, M., and Ferrari, A. Type Inference, Abstract Interpretation and Strictness Analysis. In *Theoretical Computer Science 121 (1993) 113-143.*

[2] Cousot, P., and Cousot, R. Abstract Interpretation, a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages.* 1977.

[3] Cousot, P., and Cousot, R. Inductive Definitions, Semantics and Abstract Interpretation. In *Proceedings of the 1992 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1992

[4] Dornic, V., Jouvelot, P. and Gifford, D. K. Polymorphic Time Systems for Estimating Program Complexity. *ACM Letters on Programming Languages and Systems*, vol. 1, no. 1, 1992.

[5] Jouvelot, P., and Gifford, D. K. Algebraic Reconstruction of Types and Effects. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1991.

[6] Lucassen, J. M., and Gifford, D. K. Polymorphic Effect Systems. In *Proceedings of the 1988 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1988.

[7] Mercouroff, N. *Analyse Sémantique des Communications entre Processus de Programmes Paralleles.* Ph.D. Thesis, Ecole Polytechnique, 1990.

[8] Mitchell, J. Type Systems for Programming Languages. In *Formal Models and Semantics.* Handbook of Theoretical Computer Science, Elsevier, 1990.

[9] Monsuez, B. Polymorphic Typing by Abstract Interpretation. In *Proceedings of the 12th Conference FST & TCS*, Springer Verlag LNCS 652.

[10] Mycroft, A., and Jones, N. D. A relational framework for abstract interpretation, In *Proceedings of a workshop in Copenhagen*. Ed. Ganziger and Jones, 1985, Springer Verlag LNCS 215.

[11] Nielson, H. R., and Nielson, F. Higher-Order Concurrent Programs with Finite Communication Topology, In *Proceedings of the 1994 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1994.

[12] O'Toole, J. W. Jr, and Gifford, D. K. Type Reconstruction with First-Class Polymorphic Values. In *Proceedings of the 1989 ACM Conference on Programming Language Design and Implementation*. ACM, New-York, 1989.

[13] Plotkin, G. A structural approach to operational semantics. In *Technical report DAIMI-FN-19*. Aarhus University, 1981.

[14] Reistad, B., and Gifford, D. K. Static Dependant Costs for Estimating Program Execution Time. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*. ACM, New-York, 1994.

[15] Shivers, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991.

[16] Yang, Y. M., and Jouvelot, P. Separate Abstract Interpretation for Control-Flow Analysis. In *Proceedings of the International Conference on the Theoretical Aspects of Computer Software*. LNCS 789, Springer Verlag, 1994.