Types as Abstract Interpretations

(invited paper)

Patrick Cousot
LIENS, École Normale Supérieure
45, rue d'Ulm
75230 Paris cedex 05 (France)
cousot@dmi.ens.fr, http://www.ens.fr/~cousot

Abstract

Starting from a denotational semantics of the eager untyped lambda-calculus with explicit runtime errors, the standard collecting semantics is defined as specifying the strongest program properties. By a first abstraction, a new sound type collecting semantics is derived in compositional fixpoint form. Then by successive (semi-dual) Galois connection based abstractions, type systems and/or type inference algorithms are designed as abstract semantics or abstract interpreters approximating the type collecting semantics. This leads to a hierarchy of type systems, which is part of the lattice of abstract interpretations of the untyped lambda-calculus. This hierarchy includes two new à la Church/Curry polytype systems. Abstractions of this polytype semantics lead to classical Milner/Mycroft and Damas/-Milner polymorphic type schemes, Church/Curry monotypes and Hindley principal typing algorithm. This shows that types are abstract interpretations.

1 Introduction

The leading idea of abstract interpretation [6, 7, 9, 12] is that program semantics, proof and static analysis methods have common structures which can be exhibited by abstraction of the structure of run-time computations. This leads to an organization of the more or less approximate or refined semantics into a lattice of abstract interpretations. This unifying point of view allows for a synthetic understanding of a wide range of works from theoretical semantical specifications to practical static analysis algorithms.

It will be shown that this point of view can be applied to type theory, in particular to type soundness and à la Curry type inference which, following [17, 29], have been dominating research themes in programming languages during the last two decades, at least for functional programming languages [1, 19, 31]. Traditionally the design of a type system *"involves defining the notion of type error for a given language, formalizing the type system by a set of type rules, and verifying that program execution of well-typed programs cannot produce type errors. This process, if successful, guarantees the type-soundness of a language as a whole. Type-checking algorithms can then be developed as a separate concern, and their correctness can be verified with respect to a given type system; this process guarantees that type checkers satisfy the language definition."* [2]. Abstract interpretation allows viewing all these different aspects in the more unifying framework of semantic approximation. Formalization of program analysis and type systems within the same abstract interpretation framework should lead to a better understanding of the relationship between these seemingly different approaches to program correctness and optimization.

Permission to make digital/hard copies of all or part of this material for personnal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
POPL 97, Paris, France
© 1997 ACM 0-89791-853-3/96/01 ..$3.50

2 Syntax

The syntax of the untyped eager lambda calculus is:

$$\begin{array}{rcl} \mathtt{x}, \mathtt{f}, \ldots \in \mathbb{X} & : & \text{program variables} \\ e \in \mathbb{E} & : & \text{program expressions} \\ e & ::= & \mathtt{x} \mid \boldsymbol{\lambda}\mathtt{x} \cdot e \mid e_1(e_2) \mid \boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e \mid \\ & & \mathbf{1} \mid e_1 - e_2 \mid (e_1 \,\mathbf{?}\, e_2 : e_3) \end{array}$$

$\boldsymbol{\lambda}\mathtt{x} \cdot e$ is the lambda abstraction and $e_1(e_2)$ the application. In $\boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e$, the function $\mathtt{f}$ with formal parameter $\mathtt{x}$ is defined recursively. $(e_1 \,\mathbf{?}\, e_2 : e_3)$ is the test for zero.

3 Denotational Semantics

The semantic domain $\mathbb{S}$ is defined by the following equations [20]:

$$\begin{array}{rcll} \mathbb{W} & \triangleq & \{\omega\} & \text{wrong} \\ \mathsf{z} \in \mathbb{Z} & & & \text{integers} \\ \mathsf{u}, \mathsf{f}, \varphi \in \mathbb{U} & \cong & \mathbb{W}_\perp \oplus \mathbb{Z}_\perp \oplus [\mathbb{U} \mapsto \mathbb{U}]_\perp & \text{values} \\ \mathsf{R} \in \mathbb{R} & \triangleq & \mathbb{X} \mapsto \mathbb{U} & \text{environments} \\ \phi \in \mathbb{S} & \triangleq & \mathbb{R} \mapsto \mathbb{U} & \text{semantic domain} \end{array}$$

where ω is the wrong value, $\perp$ denotes non-termination, $D_\perp$ is the lift of domain D (with up injection $\uparrow(\bullet) \in D \mapsto D_\perp$ and partial down injection $\downarrow(\bullet) \in D_\perp \rightarrowtail D$), $D_1 \oplus D_2$ is the coalesced sum of domains D_1 and D_2 (with left and right injections $\bullet :: D_1 \in D_1 \mapsto D_1 \oplus D_2$ and $\bullet :: D_2 \in D_2 \mapsto D_1 \oplus D_2$), $\Omega \triangleq \uparrow(\omega) :: \mathbb{W}_\perp$ and $[D_1 \mapsto D_2]$ is the domain of continuous, $\perp$-strict, Ω-strict functions from D_1 into D_2. $\sqsubseteq$ is the computational ordering on $\mathbb{U}$ and $\sqcup$ is the least upper bound (lub) of increasing chains.

In the metalanguage for defining the denotational semantics below, $\Lambda x^{\bullet} \ldots$ or $\Lambda x \in S^{\bullet} \ldots$ is the lambda abstraction.

$(\ldots\ ?\ \ldots\ |\ \ldots\ ?\ \ldots\ |\ \ldots)$ is the conditional expression. The assignment $\mathsf{R}[\mathsf{x}\leftarrow\mathsf{u}]$ is such that $\mathsf{R}[\mathsf{x}\leftarrow\mathsf{u}](\mathsf{x}) = \mathsf{u}$ and $\mathsf{R}[\mathsf{x}\leftarrow\mathsf{u}](\mathsf{y}) = \mathsf{R}(\mathsf{y})$ when $\mathsf{x} \neq \mathsf{y}$. $\mathbf{lfp}_{\perp}^{\sqsubseteq}\,\varphi$ is the $\sqsubseteq$-least fixpoint of the monotone operator $\varphi \in L \mapsto L$ on a complete partial order (cpo) $\langle L, \sqsubseteq\rangle$, which is greater than or equal to $\perp$ [8, 20].

The denotational semantics [19, 29]

$$\mathbf{S}[\![\bullet]\!] \in \mathbb{E} \mapsto \mathbb{S}$$

defines a call-by-value evaluation with run-time type check and propagation of errors:

$$\begin{array}{rcl}
\mathbf{S}[\![\mathsf{x}]\!] & \triangleq & \Lambda\mathsf{R}\cdot \mathsf{R}(\mathsf{x}) \\
\mathbf{S}[\![\boldsymbol{\lambda}\mathsf{x}\cdot e]\!] & \triangleq & \Lambda\mathsf{R}\cdot \uparrow\big(\Lambda\mathsf{u}\cdot(\mathsf{u} = \perp \vee \mathsf{u} = \Omega\ ?\ \mathsf{u}\ |\ \mathbf{S}[\![e]\!]\mathsf{R}[\mathsf{x}\leftarrow\mathsf{u}]\big)\big) :: [\mathbb{U} \mapsto \mathbb{U}]_{\perp} \\
\mathbf{S}[\![e_1(e_2)]\!] & \triangleq & \Lambda\mathsf{R}\cdot(\mathbf{S}[\![e_1]\!]\mathsf{R} = \perp \vee \mathbf{S}[\![e_2]\!]\mathsf{R} = \perp\ ?\ \perp\ |\ \mathbf{S}[\![e_1]\!]\mathsf{R} = \mathsf{f} :: [\mathbb{U} \mapsto \mathbb{U}]_{\perp}\ ?\ \downarrow(\mathsf{f})\big(\mathbf{S}[\![e_2]\!]\mathsf{R}\big)\ |\ \Omega) \\
\mathbf{S}[\![\boldsymbol{\mu}\mathsf{f}\cdot\boldsymbol{\lambda}\mathsf{x}\cdot e]\!] & \triangleq & \Lambda\mathsf{R}\cdot \mathbf{lfp}^{\sqsubseteq}_{\uparrow(\Lambda\mathsf{u}\cdot\perp)::[\mathbb{U}\mapsto\mathbb{U}]_{\perp}}\ \Lambda\varphi\cdot\mathbf{S}[\![\boldsymbol{\lambda}\mathsf{x}\cdot e]\!]\mathsf{R}[\mathsf{f}\leftarrow\varphi] \\
\mathbf{S}[\![\mathbf{1}]\!] & \triangleq & \Lambda\mathsf{R}\cdot \uparrow(1) :: \mathbb{Z}_{\perp} \\
\mathbf{S}[\![e_1 - e_2]\!] & \triangleq & \Lambda\mathsf{R}\cdot(\mathbf{S}[\![e_1]\!]\mathsf{R} = \perp \vee \mathbf{S}[\![e_2]\!]\mathsf{R} = \perp\ ?\ \perp\ |\ \mathbf{S}[\![e_1]\!]\mathsf{R} = \mathsf{z}_1 :: \mathbb{Z}_{\perp} \wedge \mathbf{S}[\![e_2]\!]\mathsf{R} = \mathsf{z}_2 :: \mathbb{Z}_{\perp}\ ?\ \uparrow(\downarrow(\mathsf{z}_1) - \downarrow(\mathsf{z}_2)) :: \mathbb{Z}_{\perp}\ |\ \Omega) \\
\mathbf{S}[\![(e_1\ \mathbf{?}\ e_2 : e_3)]\!] & \triangleq & \Lambda\mathsf{R}\cdot(\mathbf{S}[\![e_1]\!]\mathsf{R} = \perp\ ?\ \perp\ |\ \mathbf{S}[\![e_1]\!]\mathsf{R} = \mathsf{z} :: \mathbb{Z}_{\perp}\ ?\ (\downarrow(\mathsf{z}) = 0\ ?\ \mathbf{S}[\![e_2]\!]\mathsf{R}\ |\ \mathbf{S}[\![e_3]\!]\mathsf{R})\ |\ \Omega)
\end{array}$$

The choice of a denotational semantics instead of the usual operational semantics follows [29] and corresponds to an initial abstraction [12] where the notion of computation step is lost. The relational semantics of [30] was initially chosen but finally abandoned in favor of a semantic model where the use of fixpoints is explicit both in the standard and type semantics. The denotational semantics introduces an initial approximation which prevents discussing notions linked to computation steps such as subject reduction [31]. It illustrates the direct application of the Galois connection based abstract interpretation framework [6, 7, 9] initially conceived for transition systems, that is small-step operational semantics, to denotational semantics.

4 Standard Collecting Semantics

A semantic property is a set of possible semantics of a program. The set of semantic properties:

$$P \in \mathbb{P} \triangleq \wp(\mathbb{S})$$

is a complete boolean lattice $\langle \mathbb{P}, \subseteq, \emptyset, \mathbb{S}, \cup, \cap, \neg\rangle$ for subset inclusion $\subseteq$, that is logical implication.

The standard collecting semantics:

$$\begin{array}{l}
\mathbf{C}[\![\bullet]\!] \in \mathbb{E} \mapsto \mathbb{P} \\
\mathbf{C}[\![e]\!] \triangleq \{\mathbf{S}[\![e]\!]\}
\end{array}$$

is the strongest program property.

Type systems are understood as abstract compositional semantics of the lambda calculus which approximate the standard collecting semantics. The à la Church/Curry simple type system [1] is first considered.

5 Church/Curry Monotype Semantics

The type of a program e is a set of typings $\langle H,\ m\rangle$ stating that the standard evaluation of the program e in an environment R where global variables x have type $H(\mathsf{x})$ given by type environment H returns a value $\mathbf{S}[\![e]\!]\mathsf{R}$ with monotype m. For example, program $\boldsymbol{\lambda}\mathsf{x}\cdot\mathsf{x}$ has type $\{\langle H,\ m\mathtt{->}m\rangle \mid H \in \mathbb{H}^{\mathrm{C}} \wedge m \in \mathbb{M}^{\mathrm{C}}\}$. The syntax of types and type semantic domains are:

$m \in \mathbb{M}^{\mathrm{C}},\quad m ::= \mathtt{int} \mid m_1\mathtt{->}m_2$	monotype
$H \in \mathbb{H}^{\mathrm{C}} \triangleq \mathbb{X} \mapsto \mathbb{M}^{\mathrm{C}}$	type environment
$\theta \in \mathbb{I}^{\mathrm{C}} \triangleq \mathbb{H}^{\mathrm{C}} \times \mathbb{M}^{\mathrm{C}}$	typing
$T \in \mathbb{T}^{\mathrm{C}} \triangleq \wp(\mathbb{I}^{\mathrm{C}})$	program type

The meaning of types is defined by the *concretization function* γ^{C} as follows:

$$\begin{array}{rcl}
\gamma_1^{\mathrm{C}} & \in & \mathbb{M}^{\mathrm{C}} \mapsto \wp(\mathbb{U}) \\
\gamma_1^{\mathrm{C}}(\mathtt{int}) & \triangleq & \{\uparrow(\mathsf{z}) :: \mathbb{Z}_{\perp} \mid \mathsf{z} \in \mathbb{Z}\} \cup \{\perp\} \\
\gamma_1^{\mathrm{C}}(m_1\mathtt{->}m_2) & \triangleq & \{\uparrow(\varphi) :: [\mathbb{U} \mapsto \mathbb{U}]_{\perp} \mid \varphi \in [\mathbb{U} \mapsto \mathbb{U}] \wedge \forall \mathsf{u} \in \gamma_1^{\mathrm{C}}(m_1) : \varphi(\mathsf{u}) \in \gamma_1^{\mathrm{C}}(m_2)\} \cup \{\perp\} \\
\gamma_2^{\mathrm{C}} & \in & \mathbb{H}^{\mathrm{C}} \mapsto \wp(\mathbb{R}) \\
\gamma_2^{\mathrm{C}}(H) & \triangleq & \{\mathsf{R} \in \mathbb{R} \mid \forall \mathsf{x} \in \mathbb{X} : \mathsf{R}(\mathsf{x}) \in \gamma_1^{\mathrm{C}}(H(\mathsf{x}))\} \\
\gamma_3^{\mathrm{C}} & \in & \mathbb{I}^{\mathrm{C}} \mapsto \mathbb{P} \\
\gamma_3^{\mathrm{C}}(\langle H,\ m\rangle) & \triangleq & \{\phi \in \mathbb{S} \mid \forall \mathsf{R} \in \gamma_2^{\mathrm{C}}(H) : \phi(\mathsf{R}) \in \gamma_1^{\mathrm{C}}(m)\} \\
\gamma^{\mathrm{C}} & \in & \mathbb{T}^{\mathrm{C}} \mapsto \mathbb{P} \\
\gamma^{\mathrm{C}}(T) & \triangleq & \bigcap_{\theta \in T} \gamma_3^{\mathrm{C}}(\theta), \qquad \gamma^{\mathrm{C}}(\emptyset) \triangleq \mathbb{S}
\end{array}$$

One has:

$$\gamma^{\mathrm{C}}(\bigcup_{i\in\Delta} T_i) = \bigcap_{i\in\Delta} \gamma^{\mathrm{C}}(T_i)$$

so that there exists a unique upper adjoint $\alpha^{\mathrm{C}} \in \mathbb{P} \mapsto \mathbb{T}^{\mathrm{C}}$ such that $\langle \alpha^{\mathrm{C}},\ \gamma^{\mathrm{C}}\rangle$ is a *Galois connection* [9], a fact that is denoted:

$$\langle \mathbb{P},\ \subseteq\rangle \underset{\alpha^{\mathrm{C}}}{\overset{\gamma^{\mathrm{C}}}{\leftrightarrows}} \langle \mathbb{T}^{\mathrm{C}},\ \supseteq\rangle$$

The notation $\langle L,\ \leq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M,\ \preceq\rangle$ for semi-dual Galois connections (that are called Galois connection for short) means that $\langle L,\ \leq\rangle$ and $\langle M,\ \preceq\rangle$ are posets such that the pair of *abstraction function* $\alpha \in L \mapsto M$ and *concretization function* $\gamma \in M \mapsto L$ satisfy, for all $x \in L$ and $y \in M$:

$$\alpha(x) \preceq y \iff x \leq \gamma(y)$$

The abstraction α preserves existing lubs while the concretization γ preserves existing glbs. Reciprocally, if α preserves existing lubs or γ preserves existing glbs then there is a unique adjoint such that $\langle L,\ \leq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M,\ \preceq\rangle$ where $\alpha(x) = \mathrm{glb}\{y \mid x \leq \gamma(y)\}$ and $\gamma(y) = \mathrm{lub}\{x \mid \alpha(x) \preceq y\}$.

The intuition is that $\alpha^{\mathrm{C}}(P)$ is the best possible (i.e. most precise) type of programs with property P. This is in contrast with the use of *logical* or *algebraic relations* [24, 30, 31, 38] for which no such notion exists [1]. The critique

[1] The left image $\gamma(y) \triangleq \{x \mid \mathcal{L}(x, y)\}$ of the logical relation $\mathcal{L}$ is the usual concretization towards the standard collecting semantics so the use of a logical relation amounts to the use of a concretization only, the implicit corresponding abstraction being unused.