Types as Abstract Interpretations

Patrick COUSOT
École Normale Supérieure
DMI, 45, rue d'Ulm
75230 Paris cedex 05
France

cousot@dmi.ens.fr
http://www.ens.fr/~cousot

POPL'97, Paris, January 17, 1997

Introduction

Subject Choice

- *Twenty Years of Abstract Interpretation* would have been a nice, peaceful and restful subject;
- *Types as Abstract Interpretation* is hopefully also an interesting subject but probably more debatable and exciting:

  abstract interpretation and type theory are most often considered as separate non-interfering subjects, each with its own partisans.

Abstract Interpretation

- Abstract interpretation is a methodology for designing approximate semantics of programming languages;
- Abstract interpretation is used to soundly prove and analyze program properties [1, 2].

References

[1] P. Cousot and R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In 4th POPL, pages 238–252, Los Angeles, California, 1977. ACM Press.

[2] P. Cousot and R. Cousot. *Systematic design of program analysis frameworks*. In 6th POPL, pages 269–282, San Antonio, Texas, 1979. ACM Press.

Type Theory

- Type systems [3] and type inference [4, 5] have been a dominating research theme in programming languages in the last two decades;

References

[3] L. Damas & R. Milner. Principal type-schemes for functional programs. In 9^{th} *POPL*, pp. 207–212, Albuquerque, N.M., Jan. 1982. ACM Press.

[4] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.

[5] R. Milner. A theory of polymorphism in programming. *J. Comput. Sys. Sci.*, 17(3):348–375, Dec. 1978.

Types from an Abstract Interpretation Point of View

- Broaden the scope of abstract interpretation: is type theory an instance of abstract interpretation?
- Understand type theory from a different point of view:
  - Why is type theory so difficult?
  - Can type theory be extended to cope with more profound program semantic properties analysis?

Content of the Talk

- An abridged digest of abstract interpretation;
- Methodology of design of type systems by abstract interpretation (illustrated by Church/Curry simple monotypes);
- Application to the design of a new Church/Curry polytype system;
- Type inference and its limits in the context of program analysis.
- What is a type system?

A Digest of Abstract Interpretation

The Idea of Semantics Approximation

- Syntax;
- Standard semantics;
- Concrete properties;
- Collecting semantics;
- Abstract Properties;
- Abstraction/concretization;
- Abstract semantics.

The abstract semantics is a safe approximation of the collecting semantics.

Syntax

- The syntax defines a set of valid programs:

$e \in \mathbb{E}$ programs/expressions

Standard Semantics

- The standard semantics specifies the possible runtime behaviors of programs:

$\mathbb{S}$	semantic domain
$\mathbf{S}[\![\bullet]\!] \in \mathbb{E} \mapsto \mathbb{S}$	standard semantics

Concrete Properties

- A concrete property of a program is a set of possible program behaviors;
- The set of concrete 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.

Example of Concrete Property

- "Computing the factorial function in any environment R" is the formal property:

$$
\begin{array}{l}
\{\Lambda \mathsf{R}^\bullet\ \Lambda \mathsf{n}^\bullet\,(\,\mathsf{n} \in \mathbb{Z} \wedge 0 \leq \mathsf{n} \wedge \mathsf{n}! \leq \mathrm{maxint}\ ?\ \mathsf{n}! \\
\qquad\qquad\quad |\ \Omega\,), \\
\ \Lambda \mathsf{R}^\bullet\ \Lambda \mathsf{n}^\bullet\,(\,\mathsf{n} \in \mathbb{Z} \wedge \mathsf{n} < 0\ ?\ \bot \\
\qquad\qquad\quad |\ \mathsf{n} \in \mathbb{Z} \wedge 0 \leq \mathsf{n} \wedge \mathsf{n}! \leq \mathrm{maxint}\ ?\ \mathsf{n}! \\
\qquad\qquad\quad |\ \Omega\,)\}
\end{array}
$$

Collecting Semantics

- A collecting semantics associates a concrete property (of a given class e.g. safety, liveness, ...) to each program:

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

- The standard collecting semantics:

$$\mathbf{C}[\![e]\!] \stackrel{\Delta}{=} \{\mathbf{S}[\![e]\!]\}$$

is the strongest concrete property.

Abstract Properties

- The abstract properties correspond to a well-chosen and conveniently encoded subset of the concrete properties;
- The set of abstract properties is a complete lattice

$$\langle \mathbb{T},\ \leq,\ 0,\ 1,\ \vee,\ \wedge \rangle$$

for the approximation ordering $\leq$, corresponding to concrete subset inclusion/logical implication.

Abstraction/Concretization

- The correspondence between concrete and abstract properties is defined by a Galois connection [1]:

$$\langle \mathbb{P},\ \subseteq,\ \emptyset,\ \mathbb{S},\ \cup,\ \cap \rangle \ \underset{\alpha}{\overset{\gamma}{\leftrightarrows}}\ \langle \mathbb{T},\ \leq,\ 0,\ 1,\ \vee,\ \wedge \rangle$$

  - α: abstraction;
  - γ: concretization.

[1] For weaker models, see P. Cousot & R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, 1992.

Galois Connection

- By definition:

$$\langle \mathbb{P},\, \subseteq,\, \emptyset,\, \mathbb{S},\, \cup,\, \cap \rangle \ \underset{\alpha}{\overset{\gamma}{\leftrightarrows}}\ \langle \mathbb{T},\, \leq,\, 0,\, 1,\, \vee,\, \wedge \rangle$$

  means:

$$\forall P \in \mathbb{P} : \forall T \in \mathbb{T} : \alpha(P) \leq T \iff P \subseteq \gamma(T)$$

- The intuition is that:
  - $\alpha(P)$ is the best/strongest/most precise approximation of P;
  - $\gamma(T)$ is the meaning of T.

Abstract Semantics

- An abstract semantics associates a abstract property to each program:

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

- The abstract semantics is a safe approximation of the collecting semantics:

$$\mathbf{C}[\![e]\!] \subseteq \gamma(\mathbf{T}[\![e]\!])$$

The Abstract Interpretation Design Methodology

- Define $\mathbf{T}[\![e]\!]$ by calculation, simplifying the expression $\alpha(\mathbf{C}[\![e]\!])$, using $\leq$-approximations for simplification purposes;
- The soundness $\mathbf{S}[\![e]\!] \in \gamma(\mathbf{T}[\![e]\!])$ of the abstract semantics is by construction.

The Type Theory Design Methodology

- Syntax (for a given language);
- Standard semantics (defining type errors);
- Formalize the type system by type rules;
- Verify that execution of well-typed programs cannot produce type errors;
- Design type-checking/inference algorithms;
- Verify their correctness with respect to the type system.

Comparison of the Type Theoretic and Abstract Interpretation Design Methodologies

Thesis:

- The design of the type rules and the inference algorithm are abstract interpretations;
- The correctness criterion provides a design methodology.

Formal construction versus formal verification.

Design Methodology of a Type System by Abstract Interpretation

Simple Typing of the Eager Lambda Calculus by Abstract Interpretation

- Define the syntax, standard and collecting semantics;
- Understand the type system as an abstract semantics;
- Show that this type semantics is an abstraction of the collecting semantics (which implies that typable programs cannot go wrong);
- Show that type inference algorithms are further abstractions of the type semantics.

Syntax of the Eager Lambda Calculus

$\mathtt{x}, \mathtt{f}, \ldots \in \mathbb{X}$:	variables
$e \in \mathbb{E}$:	expressions
$e ::= \mathtt{x}$	variable
$\mid\ \boldsymbol{\lambda}\mathtt{x} \cdot e$	abstraction
$\mid\ e_1(e_2)$	application
$\mid\ \boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e$	recursion
$\mid\ \mathbf{1}$	one
$\mid\ e_1 - e_2$	difference
$\mid\ \mathbf{(}e_1 \mathbf{\,?\,} e_2 \mathbf{\,:\,} e_3\mathbf{)}$	conditional

Semantic Domains

Ω	wrong/runtime error value
$\bot$	non-termination
$\mathbb{W} \stackrel{\Delta}{=} \{\Omega\}$	wrong
$\mathsf{z} \in \mathbb{Z}$	integers
$\mathsf{u}, \mathsf{f}, \varphi \in \mathbb{U} \cong \mathbb{W}_\bot \oplus \mathbb{Z}_\bot \oplus [\mathbb{U} \mapsto \mathbb{U}]^{2}{}_\bot$	values
$\mathsf{R} \in \mathbb{R} \stackrel{\Delta}{=} \mathbb{X} \mapsto \mathbb{U}$	environments
$\phi \in \mathbb{S} \stackrel{\Delta}{=} \mathbb{R} \mapsto \mathbb{U}$	semantic domain

[2] $[\mathbb{U} \mapsto \mathbb{U}]$: continuous, $\bot$-strict, Ω-strict functions from values $\mathbb{U}$ to values $\mathbb{U}$.

Denotational Semantics of the Eager Lambda Calculus

- The denotational semantics is:

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

- The semantics $\mathbf{S}[\![1]\!]\mathsf{R}$ of constant 1 is the integer value 1 [3]:

$$\mathbf{S}[\![1]\!]\mathsf{R} \stackrel{\Delta}{=} 1$$

[3] For short up/down lifting/injection are omitted.

- The semantics $\mathbf{S}[\![e_1 - e_2]\!]\mathsf{R}$ of a difference $e_1 - e_2$:

$$\begin{aligned}\mathbf{S}[\![e_1 - e_2]\!]\mathsf{R} \stackrel{\Delta}{=} & \;(\,\mathbf{S}[\![e_1]\!]\mathsf{R} = \bot \vee \mathbf{S}[\![e_2]\!]\mathsf{R} = \bot \;?\; \bot \\ & \;|\; \mathbf{S}[\![e_1]\!]\mathsf{R} = \mathsf{z}_1 \wedge \mathbf{S}[\![e_2]\!]\mathsf{R} = \mathsf{z}_2 \;?\; \mathsf{z}_1 - \mathsf{z}_2 \\ & \;|\; \Omega\,)\end{aligned}$$

specifies that the evaluation of $e_1 - e_2$:
- does not terminate if the evaluation of e_1 or e_2 does not terminate;
- goes wrong if the evaluation of e_1 or e_2 does not return integer values;
- else the result is the difference of these values.

- The conditional $(e_1 \;\mathbf{?}\; e_2 \;\mathbf{:}\; e_3)$ is a test for zero:

$$\begin{aligned}\mathbf{S}[\![(e_1 \;\mathbf{?}\; e_2 \;\mathbf{:}\; e_3)]\!]\mathsf{R} \stackrel{\Delta}{=} & \;(\,\mathbf{S}[\![e_1]\!]\mathsf{R} = \bot \;?\; \bot \\ & \;|\; \mathbf{S}[\![e_1]\!]\mathsf{R} = 0 \;?\; \mathbf{S}[\![e_2]\!]\mathsf{R} \\ & \;|\; \mathbf{S}[\![e_1]\!]\mathsf{R} = \mathsf{z} \neq 0 \;?\; \mathbf{S}[\![e_3]\!]\mathsf{R} \\ & \;|\; \Omega\,)\end{aligned}$$

- the evaluation does not terminate if e_1 does not terminate;
- the evaluation goes wrong if e_1 does not return an integer value;
- if e_1 is 0 then the result is the value of e_2 else that of e_3.

- The semantics $\mathbf{S}[\![\mathtt{x}]\!]\mathsf{R}$ of variable $\mathtt{x}$ in environment R is the value $\mathsf{R}(\mathtt{x})$ of $\mathtt{x}$ in R:

$$\mathbf{S}[\![\mathtt{x}]\!]\mathsf{R} \stackrel{\triangle}{=} \mathsf{R}(\mathtt{x})$$

- The semantics $\mathbf{S}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!]\mathsf{R}$ of an abstraction $\boldsymbol{\lambda}\mathtt{x}\cdot e$ in environment R is a $\bot$-strict, Ω-strict continuous function:

$$\begin{aligned}\mathbf{S}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!]\mathsf{R} \stackrel{\triangle}{=} \Lambda\mathsf{u}\bullet (\, & \mathsf{u} = \bot \;?\; \bot \\ \mid\, & \mathsf{u} = \Omega \;?\; \Omega \\ \mid\, & \mathbf{S}[\![e]\!]\mathsf{R}[\mathtt{x}\leftarrow\mathsf{u}]\,)\end{aligned}$$

It maps the value u of the parameter $\mathtt{x}$ to the value $\mathbf{S}[\![e]\!]\mathsf{R}[\mathtt{x}\leftarrow\mathsf{u}]$ of the body e in the environment $\mathsf{R}[\mathtt{x}\leftarrow\mathsf{u}]$ which is R where $\mathtt{x}$ has value u.

- The semantics $\mathbf{S}[\![e_1(e_2)]\!]\mathsf{R}$ of an application $e_1(e_2)$:

$$\begin{aligned}\mathbf{S}[\![e_1(e_2)]\!]\mathsf{R} \stackrel{\triangle}{=} (\, & \mathbf{S}[\![e_1]\!]\mathsf{R} = \bot \vee \mathbf{S}[\![e_2]\!]\mathsf{R} = \bot \;?\; \bot \\ \mid\, & \mathbf{S}[\![e_1]\!]\mathsf{R} = \mathsf{f} \in [\mathbb{U} \mapsto \mathbb{U}] \;?\; \mathsf{f}\big(\mathbf{S}[\![e_2]\!]\mathsf{R}\big) \\ \mid\, & \Omega\,)\end{aligned}$$

specifies that:

- the application does not terminate if the evaluation of e_1 or e_2 does not terminate;
- e_1 should evaluate to a function $\mathsf{f} = \mathbf{S}[\![e_1]\!]\mathsf{R}$ [4] and the result is the application of f to the value $\mathbf{S}[\![e_2]\!]\mathsf{R}$ of e_2 [5]

[4] else evaluation goes wrong.

[5] The result may be $\bot$ in case of non-termination of the call or Ω if it goes wrong.

- A recursive definition $\boldsymbol{\mu}\mathtt{f}\cdot\boldsymbol{\lambda}\mathtt{x}\cdot e$ defines a function φ as the abstraction $\boldsymbol{\lambda}\mathtt{x}\cdot e$ where every occurrence of variable $\mathtt{f}$ within the body e refers to φ:

$$\mathbf{S}[\![\boldsymbol{\mu}\mathtt{f}\cdot\boldsymbol{\lambda}\mathtt{x}\cdot e]\!]\mathsf{R} \stackrel{\triangle}{=} \mathbf{lfp}^{\sqsubseteq}\ \Lambda\varphi\bullet\ \mathbf{S}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!]\mathsf{R}[\mathtt{f}\leftarrow\varphi]$$

The choice of a least fixpoint for Scott-ordering $\sqsubseteq$ ensures that no result can be returned before the computation ends.

- The let construct is defined such that:

$$\mathbf{S}[\![\texttt{let}\ \mathtt{x} = e_1\ \texttt{in}\ e_2]\!] \stackrel{\triangle}{=} \mathbf{S}[\![(\boldsymbol{\lambda}\mathtt{x} \cdot e_2)(e_1)]\!]$$

Note: e_1 is evaluated even when x is not used in e_2 (call by value).

Standard Semantics

- This denotational semantics is chosen as the standard semantics specifying run-time program behaviors;
- Important characteristics:
  - *functional* presentation, explicit *fixpoints*,
  - explicit handling of *nontermination*,
  - the semantics specifies a *run-time/dynamic type checking*;
- Other semantics would only require further refinements/abstractions.

Standard Collecting Semantics

- The standard collecting semantics:

$$\mathbf{C}[\![\bullet]\!] \in \mathbb{E} \mapsto \mathbb{P}$$
$$\mathbf{C}[\![e]\!] \stackrel{\triangle}{=} \{\mathbf{S}[\![e]\!]\}$$

is the strongest concrete property.

Church/Curry Monotypes

- Simple types are monomorphic:

$$m \in \mathbb{M}^c, \quad m ::= \texttt{int} \mid m_1 \to m_2 \qquad \text{monotype}$$

- A type environment associates a type to free program variables:

$$H \in \mathbb{H}^c \stackrel{\triangle}{=} \mathbb{X} \mapsto \mathbb{M}^c \qquad \text{type environment}$$

Church/Curry Monotypes (continued)

- A typing $\langle H, m\rangle$ specifies a possible result type m in a given type environment H assigning types to free variables:

$$\theta \in \mathbb{I}^{c} \triangleq \mathbb{H}^{c} \times \mathbb{M}^{c} \qquad \text{typing}$$

- An abstract property or program type is a set of typings;

$$T \in \mathbb{T}^{c} \triangleq \wp(\mathbb{I}^{c}) \qquad \text{program type}$$

Concretization Function

The meaning of types is a program property, as defined by the concretization function γ^{c}: [6]

- Monotypes $\gamma_1^{c} \in \mathbb{M}^{c} \mapsto \wp(\mathbb{U})$:

$$\gamma_1^{c}(\mathtt{int}) \triangleq \mathbb{Z} \cup \{\perp\}$$

$$\gamma_1^{c}(m_1 \mathbin{\text{->}} m_2) \triangleq \{\varphi \in [\mathbb{U} \mapsto \mathbb{U}] \mid \forall \mathsf{u} \in \gamma_1^{c}(m_1) : \varphi(\mathsf{u}) \in \gamma_1^{c}(m_2)\} \cup \{\perp\}$$

[6] For short up/down lifting/injection are omitted.

- type environment $\gamma_2^{c} \in \mathbb{H}^{c} \mapsto \wp(\mathbb{R})$:

$$\gamma_2^{c}(H) \triangleq \{\mathsf{R} \in \mathbb{R} \mid \forall \mathbf{x} \in \mathbb{X} : \mathsf{R}(\mathbf{x}) \in \gamma_1^{c}(H(\mathbf{x}))\}$$

- typing $\gamma_3^{c} \in \mathbb{I}^{c} \mapsto \mathbb{P}$:

$$\gamma_3^{c}(\langle H,\ m\rangle) \triangleq \{\phi \in \mathbb{S} \mid \forall \mathsf{R} \in \gamma_2^{c}(H) : \phi(\mathsf{R}) \in \gamma_1^{c}(m)\}$$

- program type $\gamma^{c} \in \mathbb{T}^{c} \mapsto \mathbb{P}$:

$$\gamma^{c}(T) \triangleq \bigcap_{\theta \in T} \gamma_3^{c}(\theta)$$

$$\gamma^{c}(\emptyset) \triangleq \mathbb{S}$$

- Types exclude going wrong:

$$\Omega \in \gamma^{c}(T) \iff T = \emptyset$$

Galois Connection

- Disjunction of properties correspond to intersection of types

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

so that the correspondence between concrete properties and program types is a Galois connection:

$$\langle \mathbb{P}, \subseteq, \emptyset, \mathbb{S}, \cup, \cap\rangle \underset{\alpha^c}{\overset{\gamma^c}{\leftrightarrows}} \langle \mathbb{T}^c, \supseteq, \mathbb{I}^c, \emptyset, \cap, \cup\rangle$$

Church/Curry Monotyping Rules

$$H \vdash^{c} \mathtt{x} \Rightarrow H(\mathtt{x}) \qquad \text{(VAR)}$$

$$\frac{H \vdash^{c} e_1 \Rightarrow m_1 \to m_2, \quad H \vdash^{c} e_2 \Rightarrow m_1}{H \vdash^{c} e_1(e_2) \Rightarrow m_2} \qquad \text{(APP)}$$

$$\frac{H[\mathtt{x} \leftarrow m_1] \vdash^{c} e \Rightarrow m_2}{H \vdash^{c} \boldsymbol{\lambda}\mathtt{x} \cdot e \Rightarrow m_1 \to m_2} \qquad \text{(ABS)}$$

$$\frac{H[\mathtt{f} \leftarrow m] \vdash^{c} \boldsymbol{\lambda}\mathtt{x} \cdot e \Rightarrow m}{H \vdash^{c} \boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e \Rightarrow m} \qquad \text{(REC)}$$

$$H \vdash^{c} \mathbf{1} \Rightarrow \mathtt{int} \qquad \text{(CST)}$$

$$\frac{H \vdash^{c} e_1 \Rightarrow \mathtt{int}, \quad H \vdash^{c} e_2 \Rightarrow \mathtt{int}}{H \vdash^{c} e_1 - e_2 \Rightarrow \mathtt{int}} \qquad \text{(DIF)}$$

$$\frac{H \vdash^{c} e_1 \Rightarrow \mathtt{int}, \quad H \vdash^{c} e_2 \Rightarrow m, \quad H \vdash^{c} e_3 \Rightarrow m}{H \vdash^{c} (e_1 \mathbin{\mathbf{?}} e_2 \mathbin{\mathbf{:}} e_3) \Rightarrow m} \qquad \text{(CND)}$$

Abstract Semantics

- These typing rules can be understood as a compositional abstract semantics:
  $$\mathbf{T}[\![\bullet]\!] \in \mathbb{E} \mapsto \mathbb{T}^c$$
- Reciprocally, a compositional abstract semantics can be presented using inference rules;
- This follows from a general result [6], showing equivalence of various semantics presentations.

Reference

[6] P. Cousot & R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, invited paper. CAV '95, LNCS 939, pp. 293–308, 1995.

Church/Curry Monotype Semantics

$$\mathbf{T}^{c}[\![\mathtt{x}]\!] \triangleq \{\langle H,\ H(\mathtt{x})\rangle \mid H \in \mathbb{H}^{c}\} \qquad \text{(VAR)}$$

$$\mathbf{T}^{c}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!] \triangleq \{\langle H,\ m_1 \rightarrow m_2\rangle \mid \langle H[\mathtt{x} \leftarrow m_1],\ m_2\rangle \in \mathbf{T}^{c}[\![e]\!]\} \qquad \text{(ABS)}$$

$$\mathbf{T}^{c}[\![e_1(e_2)]\!] \triangleq \{\langle H,\ m_2\rangle \mid \langle H,\ m_1 \rightarrow m_2\rangle \in \mathbf{T}^{c}[\![e_1]\!] \wedge \langle H,\ m_1\rangle \in \mathbf{T}^{c}[\![e_2]\!]\} \qquad \text{(APP)}$$

$$\mathbf{T}^{c}[\![\boldsymbol{\mu}\mathtt{f}\cdot\boldsymbol{\lambda}\mathtt{x}\cdot e]\!] \triangleq \{\langle H,\ m\rangle \mid \langle H[\mathtt{f} \leftarrow m],\ m\rangle \in \mathbf{T}^{c}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!]\} \qquad \text{(REC)}$$

$$\mathbf{T}^{c}[\![1]\!] \triangleq \{\langle H,\ \mathtt{int}\rangle \mid H \in \mathbb{H}^{c}\} \qquad \text{(CST)}$$

$$\mathbf{T}^{c}[\![e_1 - e_2]\!] \triangleq \{\langle H,\ \mathtt{int}\rangle \mid \langle H,\ \mathtt{int}\rangle \in \mathbf{T}^{c}[\![e_1]\!] \cap \mathbf{T}^{c}[\![e_2]\!]\} \qquad \text{(DIF)}$$

$$\mathbf{T}^{c}[\![(e_1\ \mathbf{?}\ e_2\ \mathbf{:}\ e_3)]\!] \triangleq \{\langle H,\ m\rangle \mid \langle H,\ \mathtt{int}\rangle \in \mathbf{T}^{c}[\![e_1]\!] \wedge \langle H,\ m\rangle \in \mathbf{T}^{c}[\![e_2]\!] \cap \mathbf{T}^{c}[\![e_3]\!]\} \qquad \text{(CND)}$$

Abstraction Theorem

- The Church/Curry type semantics $\mathbf{T}^{c}[\![\bullet]\!]$ is an upper-approximation[7] of the strongest collecting semantics (for the standard denotational semantics):

$$\begin{array}{rl} & \alpha^{c}(\mathbf{C}[\![e]\!]) \supseteq \mathbf{T}^{c}[\![e]\!] \\ \Longleftrightarrow & \mathbf{C}[\![e]\!] \subseteq \gamma^{c}(\mathbf{T}^{c}[\![e]\!]) \\ \Longleftrightarrow & \mathbf{S}[\![e]\!] \in \gamma^{c}(\mathbf{T}^{c}[\![e]\!]) \end{array}$$

[7] $\subseteq$-upper concrete and $\supseteq$-upper (i.e. $\subseteq$-lower) abstract approximation

Typable Programs Cannot Go Wrong

- A program e is typable iff $\mathbf{T}^{c}[\![e]\!] \neq \emptyset$.
- Typable programs cannot go wrong:

$$\langle H,\ m\rangle \in \mathbf{T}^{c}[\![e]\!] \wedge \mathsf{R} \in \gamma^{c}_{2}(H) \Longrightarrow \mathbf{S}[\![e]\!]\mathsf{R} \neq \Omega$$

Design Methodology

- The soundness requirement that the abstract semantics is an $\supseteq$-upper abstract approximation of the concrete/collecting semantics:

$$\begin{array}{rl} & \mathbf{C}[\![e]\!] \subseteq \gamma^{\mathsf{c}}(\mathbf{T}^{\mathsf{c}}[\![e]\!]) \\ \Longleftrightarrow & \alpha^{\mathsf{c}}(\mathbf{C}[\![e]\!]) \supseteq \mathbf{T}^{\mathsf{c}}[\![e]\!] \end{array}$$

provides a design methodology:

Design $\mathbf{T}^{\mathsf{c}}[\![e]\!]$ by calculation, starting from the best possible choice: $\alpha^{\mathsf{c}}(\mathbf{C}[\![e]\!])$.

A Simplistic Example . . . Constants

$$\begin{array}{ll} \mathbf{T}^{\mathsf{c}}[\![1]\!] & \\ \triangleq \alpha^{\mathsf{c}}(\mathbf{C}[\![1]\!]) & \text{best possible choice} \\ = \alpha^{\mathsf{c}}(\{\mathbf{S}[\![1]\!]\}) & \text{def. collecting sem.} \\ = \alpha^{\mathsf{c}}(\{\Lambda \mathsf{R}^{\bullet}\, 1\}) & \text{def. standard sem.} \\ = \bigcup\{T \mid \{\Lambda \mathsf{R}^{\bullet}\, 1\} \subseteq \gamma^{\mathsf{c}}(T)\} & \text{Galois connection} \\ = \bigcup\{T \mid \forall \theta \in T : \Lambda \mathsf{R}^{\bullet}\, 1 \in \gamma_3^{\mathsf{c}}(\theta)\} & \text{def. } \gamma^{\mathsf{c}} \\ = \{\theta \mid \Lambda \mathsf{R}^{\bullet}\, 1 \in \gamma_3^{\mathsf{c}}(\theta)\} & \text{set theory} \\ = \{\langle H,\, m\rangle \mid \forall \mathsf{R} \in \gamma_2^{\mathsf{c}}(H) : 1 \in \gamma_1^{\mathsf{c}}(m)\} & \text{def. } \gamma_3^{\mathsf{c}} \\ = \{\langle H,\, \mathtt{int}\rangle \mid H \in \mathbb{H}^{\mathsf{c}}\} & \text{def. } \gamma_1^{\mathsf{c}} \end{array}$$

Design of a New Church/Curry Polytype System by Abstract Interpretation

A New Type System

- The methodology is illustrated by the design of a new simple polytype system:

  the Church/Curry polytype system:

  - polytypes are infinitary conjunctions of monotypes;
  - `let`-polymorphism, à la Milner;
  - more powerful than the Damas-Miner and Milner-Mycroft polymorphic type systems;
  - no complete inference algorithm.

Church/Curry Polytypes

$m \in \mathbb{M}^{\text{PC}}, \quad m ::= \texttt{int} \mid m_1 \to m_2$	monotype
$p \in \mathbb{P}^{\text{PC}} \triangleq \wp(\mathbb{M}^{\text{PC}})$	polytype
$H \in \mathbb{H}^{\text{PC}} \triangleq \mathbb{X} \mapsto \mathbb{P}^{\text{PC}}$	type environment
$\theta \in \mathbb{I}^{\text{PC}} \triangleq \mathbb{H}^{\text{PC}} \times \mathbb{M}^{\text{PC}}$	typing
$T \in \mathbb{T}^{\text{PC}} \triangleq \wp(\mathbb{I}^{\text{PC}})$	program type

Polymorphism is restricted to environments.

Galois Connection

- The correspondence between polytypes and program properties is a Galois connection:

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

- Concretization of monotypes $\gamma_1^{\text{PC}} \in \mathbb{M}^{\text{PC}} \mapsto \wp(\mathbb{U})$ (unchanged):

$$\gamma_1^{\text{PC}}(\texttt{int}) \triangleq \mathbb{Z} \cup \{\bot\}$$
$$\gamma_1^{\text{PC}}(m_1 \to m_2) \triangleq \{\varphi \in [\mathbb{U} \mapsto \mathbb{U}] \mid \forall \mathsf{u} \in \gamma_1^{\text{PC}}(m_1) : \varphi(\mathsf{u}) \in \gamma_1^{\text{PC}}(m_2)\} \cup \{\bot\}$$

- Concretization of polytypes $\gamma_2^{\text{PC}} \in \mathbb{P}^{\text{PC}} \mapsto \wp(\mathbb{U})$:

$$\gamma_2^{\text{PC}}(p) \triangleq \bigcap_{m \in p} \gamma_1^{\text{PC}}(m)$$
$$\gamma_2^{\text{PC}}(\emptyset) \triangleq \mathbb{U}$$

- Concretization of polytype environments $\gamma_3^{\text{PC}} \in \mathbb{H}^{\text{PC}} \mapsto \wp(\mathbb{R})$:

$$\gamma_3^{\text{PC}}(H) \triangleq \{\mathsf{R} \in \mathbb{R} \mid \forall \mathbf{x} \in \mathbb{X} : \mathsf{R}(\mathbf{x}) \in \gamma_2^{\text{PC}}(H(\mathbf{x}))\}$$

- Concretization of polytypings $\gamma_4^{\text{PC}} \in \mathbb{I}^{\text{PC}} \mapsto \mathbb{P}$:

$$\gamma_4^{\text{PC}}(\langle H, m \rangle) \triangleq \{\phi \in \mathbb{S} \mid \forall \mathsf{R} \in \gamma_3^{\text{PC}}(H) : \phi(\mathsf{R}) \in \gamma_1^{\text{PC}}(m)\}$$

- Concretization of program types $\gamma^{\text{PC}} \in \mathbb{T}^{\text{PC}} \mapsto \mathbb{P}$:

$$\gamma^{\text{PC}}(T) \triangleq \bigcap_{\theta \in T} \gamma_4^{\text{PC}}(\theta)$$
$$\gamma^{\text{PC}}(\emptyset) \triangleq \mathbb{S}$$

- If $T \neq \emptyset$ then $\gamma^{\text{PC}}(T)$ excludes semantical values going wrong.

Church/Curry Polytype Semantics

- The Church/Curry polytype semantics:

$$\mathbf{T}^{\text{PC}}[\![\bullet]\!] \in \mathbb{E} \mapsto \mathbb{T}^{\text{PC}}$$

is designed according to the soundness condition:

$$\begin{array}{ll} & \alpha^{\text{PC}}(\mathbf{C}[\![e]\!]) \supseteq \mathbf{T}^{\text{PC}}[\![e]\!] \\ \Longleftrightarrow & \mathbf{C}[\![e]\!] \subseteq \gamma^{\text{PC}}(\mathbf{T}^{\text{PC}}[\![e]\!]) \\ \Longleftrightarrow & \mathbf{S}[\![e]\!] \in \gamma^{\text{PC}}(\mathbf{T}^{\text{PC}}[\![e]\!]) \end{array}$$

so that typable programs cannot go wrong;

- By calculation we derive a compositional functional fixpoint definition of $\mathbf{T}^{\text{PC}}[\![\bullet]\!]$;
- We then express this polytype semantics in equivalent rule-based form.

Church/Curry Polytyping Rules

- Rule-based presentation of the polytype semantics $\mathbf{T}^{\text{PC}}[\![\bullet]\!]$:

$$\frac{m \in H(\mathtt{x})}{H \vdash^{\text{PC}} \mathtt{x} \Rightarrow m} \qquad \text{(VAR)}$$

$$\frac{H[\mathtt{x} \leftarrow \{m_1\}] \vdash^{\text{PC}} e \Rightarrow m_2}{H \vdash^{\text{PC}} \boldsymbol{\lambda}\mathtt{x} \cdot e \Rightarrow m_1 \rightarrow m_2} \qquad \text{(APP)}$$

$$\frac{H \vdash^{\text{PC}} e_1 \Rightarrow m_1 \rightarrow m_2, \;\; H \vdash^{\text{PC}} e_2 \Rightarrow m_1}{H \vdash^{\text{PC}} e_1(e_2) \Rightarrow m_2} \qquad \text{(ABS)}$$

$$\frac{\begin{array}{c} p_1 \neq \emptyset, \;\; \forall m_1 \in p_1 : H \vdash^{\text{PC}} e_1 \Rightarrow m_1, \\ H[\mathtt{x} \leftarrow p_1] \vdash^{\text{PC}} e_2 \Rightarrow m_2 \end{array}}{H \vdash^{\text{PC}} \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2 \Rightarrow m_2} \qquad \text{(LET)}$$

$$\frac{\forall m_1 \in p_1 : H[\mathtt{f} \leftarrow p_1] \vdash^{\text{PC}} \boldsymbol{\lambda}\mathtt{x} \cdot e \Rightarrow m_1, \;\; m \in p_1}{H \vdash^{\text{PC}} \boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e \Rightarrow m} \qquad \text{(REC)}$$

Church/Curry Polytyping Semantics

The most interesting case is for recursion:

$$\mathbf{T}^{\text{PC}}[\![\boldsymbol{\mu}\mathtt{f}\cdot\boldsymbol{\lambda}\mathtt{x}\cdot e]\!] \triangleq \{\langle H,\ m\rangle \mid m \in \mathbf{gfp}^{\subseteq}_{\mathbb{M}^{\text{PC}}\to\mathbb{M}^{\text{PC}}}\ \Psi\}$$

where $\Psi \triangleq \Lambda p^{\bullet}\{m' \mid \langle H[\mathtt{f}\leftarrow p],\ m'\rangle \in \mathbf{T}^{\text{PC}}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!]\}$

or equivalently:

$$\begin{aligned}\mathbf{T}^{\text{PC}}[\![\boldsymbol{\mu}\mathtt{f}\cdot\boldsymbol{\lambda}\mathtt{x}\cdot e]\!] = \{\langle H,\ m\rangle \mid\ & \exists p \subseteq \mathbb{M}^{\text{PC}}\to\mathbb{M}^{\text{PC}} : m \in p \wedge \\ & \forall m' \in p : \langle H[\mathtt{f}\leftarrow p],\ m'\rangle \in \mathbf{T}^{\text{PC}}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!]\}\end{aligned}$$

A Typable Program ...

The ML program:

```
let rec F f g n x =
  if n = 0 then g(x)
  else F(f)(fun x ->(fun h -> g(h(x))))(n-1)(x)(f);;
```

such that:

$$\mathtt{F\ f\ g\ n\ x} = \mathtt{g}(\mathtt{f}^{\mathtt{n}}(\mathtt{x})))$$

has type:

$$\{\langle H,\ (m_1 \to m_1) \to ((m_1 \to m_2) \to (\mathtt{int} \to (m_1 \to m_2)))\rangle \mid H \in \mathbb{H}^{\text{PC}} \wedge m_1, m_2 \in \mathbb{M}^{\text{PC}}\}$$

... without Hindley/Milner Type

```
>       Caml Light version 0.71/mac
#let rec F f g n x =
   if n = 0 then g(x)
   else F(f)(fun x -> (fun h -> g(h(x))))(n-1)(x)(f);;
Toplevel input:
>.......F f g n x =
>   if n = 0 then g(x)
>   else F(f)(fun x -> (fun h -> g(h(x))))(n-1)(x)(f)..
This expression has type
 'a -> ('b -> 'c) -> int -> 'b -> 'c,
but is used with type
 'a -> ('b -> ('b -> 'b) -> 'c) -> int -> 'b -> 'a -> 'c.
```


What is the Problem?

- À la Milner typing makes rough program-independent approximations of fixpoints;
- The abstract interpretation iteration strategy for fixpoints is more refined [8],
- Worst, à la Milner typing uses specific type property and language-dependent abstractions which are not general enough for program analysis [9].

[8] convergence may have to be enforced through with widenings & narrowings
[9] Mostly applicable to boolean abstract domains.

Type Inference and its Limits for Program Analysis

Herbrand Abstraction

- The Herbrand abstraction **lcg** (least common generalization) can be used to abstract an (infinite) set of ground terms by a single, machine-representable term with variables.
- For example, up to variable `'a`, `'b`, ... renaming:

$$\mathbf{lcg}(\{(m_1 \to m_1) \to ((m_1 \to m_2) \to (\mathtt{int} \to (m_1 \to m_2))) \mid m_1, m_2 \in \mathbb{M}^{\scriptscriptstyle PC}\})$$

$=$ `('a -> 'a) -> (('a -> 'b) -> (int ->('a -> 'b)))`

Monotypes with Variables

$\text{'a} \in \mathbb{V}$	type variables
$\tau \in \mathbb{M}^{\scriptscriptstyle H}_{\scriptscriptstyle V}$	monotype with variables
$\tau ::= \mathtt{int} \mid \text{'a} \mid \tau_1 \to \tau_2$	
$H \in \mathbb{H}^{\scriptscriptstyle H} \stackrel{\triangle}{=} \mathbb{X} \mapsto \mathbb{M}^{\scriptscriptstyle H}_{\scriptscriptstyle V}$	type environment
$T \in \mathbb{T}^{\scriptscriptstyle H} \stackrel{\triangle}{=} \mathbb{H}^{\scriptscriptstyle H} \times \mathbb{M}^{\scriptscriptstyle H}_{\scriptscriptstyle V}$	program typing

Concretization

$$\begin{aligned}
\mathbf{ground}(\emptyset) &= \emptyset \\
\mathbf{ground}(\mathtt{int}) &= \{\mathtt{int}\} \\
\mathbf{ground}(\text{'a} \to \text{'a}) &= \{m \to m \mid m \in \mathbb{M}^{\scriptscriptstyle C}\} \\
\mathbf{ground}(\text{'a}) &= \mathbb{M}^{\scriptscriptstyle C}
\end{aligned}$$

Galois Connection

- The Herbrand abstraction $\mathbf{lcg}$ is a Galois connection:

$$\langle \wp(\mathbf{ground}(T)),\ \subseteq,\ \emptyset,\ \mathbf{ground}(T),\ \cup,\ \cap\rangle$$
$$\underset{\mathbf{lcg}}{\overset{\mathbf{ground}}{\leftrightarrows}}\ \langle T^{\emptyset}/_{\equiv},\ \leq,\ \emptyset,\ [\text{'a}]_{\equiv},\ \mathbf{lcg},\ \mathbf{gci}\rangle$$

where:
  - T: set of terms with variables 'a, …,
  - $\mathbf{lcg}$: least common generalization,
  - $\mathbf{ground}$: set of ground instances,
  - $\leq$: instance preordering,
  - $\mathbf{gci}$: greatest common instance.

Imprecision

- The Herbrand abstraction $\mathbf{lcg}$ approximates a set of monotypes by a single monotype with variables;
- The Herbrand abstraction $\mathbf{lcg}$ is quite approximate:

$$\mathbf{lcg}(\{\mathtt{int}, \mathtt{int} \rightarrow \mathtt{int}\}) = \mathtt{'a}$$

- This e.g. excludes an overloaded primitive f which would behave has an `int->int` function in a call context and as an `int` (e.g. $f(0)$) in the context of an arithmetic operand.

Wrong Direction of Approximation

- The Church/Curry monotype abstraction:

$$\langle \mathbb{P},\ \subseteq,\ \emptyset,\ \mathbb{S},\ \cup,\ \cap\rangle\ \underset{\alpha^{c}}{\overset{\gamma^{c}}{\leftrightarrows}}\ \langle \mathbb{T}^{c},\ \supseteq,\ \mathbb{I}^{c},\ \emptyset,\ \cap,\ \cup\rangle$$

- The Herbrand abstraction:

$$\langle \mathbb{T}^{c},\ \subseteq,\ \emptyset,\ \mathbb{I}^{c},\ \cup,\ \cap\rangle\ \underset{\mathbf{lcg}}{\overset{\mathbf{ground}}{\leftrightarrows}}\ \langle \mathbb{T}^{\mathrm{H}\,\emptyset}/_{\equiv},\ \leq,\ \emptyset,\ [\alpha]_{\equiv},\ \mathbf{lcg},\ \mathbf{gci}\rangle$$

They do not compose!

A Specific Solution: Exact Herbrand Abstraction

- For soundness, we need a $\supseteq$-upper (i.e. $\subseteq$-lower) approximation of sets of monotypes;
- The Herbrand abstraction provides a $\subseteq$-upper approximation by a monotype with variable;
- The specific solution is to require equality, just for those sets of monotypes considered in the Church/Curry monotype semantics;
- This can always been obtained by restricting the considered types and language!

Exactness

We have:

$$\mathbf{T}^{\mathrm{c}}[\![\bullet]\!] \in \mathbb{E} \mapsto \langle \mathbb{T}^{\mathrm{c}},\ \supseteq\rangle$$

$$\langle \mathbb{T}^{\mathrm{c}},\ \subseteq\rangle \quad \underset{\mathbf{lcg}}{\overset{\mathbf{ground}}{\leftrightarrows}} \quad \langle {\mathbb{T}^{\mathrm{H}}}^{\emptyset}/_{\equiv},\ \leq\rangle$$

We design, by calculation:

$$\mathbf{T}^{\mathrm{H}}[\![e]\!] = \mathbf{lcg}(\mathbf{T}^{\mathrm{c}}[\![e]\!])$$

and, for soundness, require:

$$\mathbf{ground}(\mathbf{T}^{\mathrm{H}}[\![e]\!]) = \mathbf{T}^{\mathrm{c}}[\![e]\!]$$

Exact Abstract Semantics

- It follows that Church/Curry monotype semantics $\mathbf{T}^{\mathrm{c}}[\![\bullet]\!]$ and Hindley monotype with variables semantics $\mathbf{T}^{\mathrm{H}}[\![\bullet]\!]$ are $\langle\mathbf{lcg},\ \mathbf{ground}\rangle$ isomorphic;
- For other points of $\mathbb{T}^{\mathrm{c}}$, outside $\{\mathbf{T}^{\mathrm{c}}[\![e]\!] \mid e \in \mathbb{E}\}$, the approximation may be unsound;
- So the language $\mathbb{E}$ has to be somewhat singular, since the slightest change in the standard semantics might be unsound.

Design of the Type Semantics/Inference Algorithm

- Define $\mathbf{T}^{\mathrm{H}}[\![e]\!]$ by calculation of the expression $\alpha^{\mathrm{H}}(\mathbf{T}^{\mathrm{c}}[\![e]\!])$ (α^{H} based on $\mathbf{lcg}$);
- Check that $\gamma^{\mathrm{H}}(\mathbf{T}^{\mathrm{H}}[\![e]\!]) = \mathbf{T}^{\mathrm{c}}[\![e]\!]$ to ensure soundness (γ^{H} based on $\mathbf{ground}$);
- If the elements of the abstract domain are computer-representable and the abstract semantics is computable then the abstract semantics is a specification of a type inference algorithm.

Hindley Monotype Semantics

$$\mathbf{T}^{\mathrm{H}}[\![\mathtt{x}]\!] \triangleq \langle \mathcal{H},\ \mathcal{H}(\mathtt{x})\rangle$$

$$\mathbf{T}^{\mathrm{H}}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!] \triangleq (\ \mathbf{T}^{\mathrm{H}}[\![e]\!] = \langle H,\ \tau\rangle\ ?\ \langle H[\mathtt{x} \leftarrow {}'\mathtt{a}],\ H(\mathtt{x}) \rightarrow \tau\rangle \mid \emptyset\ )$$

$$\mathbf{T}^{\mathrm{H}}[\![e_1(e_2)]\!] \triangleq (\ \mathbf{T}^{\mathrm{H}}[\![e_2]\!] = \langle H_2, \tau_2\rangle \wedge \mathbf{gci}\{\mathbf{T}^{\mathrm{H}}[\![e_1]\!], \langle H_2, \tau_2 \rightarrow {}'\mathtt{a}\rangle\} = \langle H,\ \tau_2 \rightarrow \tau\rangle\ ?\ \langle H,\ \tau\rangle \mid \emptyset\ )$$

$$\mathbf{T}^{\mathrm{H}}[\![\boldsymbol{\mu}\mathtt{f}\cdot\boldsymbol{\lambda}\mathtt{x}\cdot e]\!] \triangleq (\ \mathbf{T}^{\mathrm{H}}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!] = \langle H,\ \tau\rangle \wedge \sigma = \mathbf{mgu}\{{}'\mathtt{a} \rightarrow {}'\mathtt{b}, H(\mathtt{f}), \tau\} \neq \emptyset\ ?\ \langle \sigma(H)[\mathtt{f} \leftarrow {}'\mathtt{c}],\ \sigma(\tau)\rangle \mid \emptyset\ )$$

$$\mathbf{T}^{\mathrm{H}}[\![\mathbf{1}]\!] \stackrel{\triangle}{=} \langle \mathcal{H},\ \mathtt{int} \rangle$$

$$\mathbf{T}^{\mathrm{H}}[\![e_1 - e_2]\!] \stackrel{\triangle}{=} \mathbf{gci}\{\langle \mathcal{H},\ \mathtt{int} \rangle, \mathbf{T}^{\mathrm{H}}[\![e_1]\!], \mathbf{T}^{\mathrm{H}}[\![e_2]\!]\}$$

$$\mathbf{T}^{\mathrm{H}}[\![\mathbf{(}e_1 \mathbf{\ ?\ } e_2 \mathbf{\ :\ } e_3\mathbf{)}]\!] \stackrel{\triangle}{=} (\ \mathbf{T}^{\mathrm{H}}[\![e_1]\!] = \langle H_1,\ \mathtt{int} \rangle \ ? \\ \mathbf{gci}\{\langle H_1,\ \mathbf{'a} \rangle, \mathbf{T}^{\mathrm{H}}[\![e_2]\!], \mathbf{T}^{\mathrm{H}}[\![e_3]\!]\} \mid \emptyset\ )$$

Limits of the Exact Herbrand Abstraction

- Language dependent (e.g. introduction of an $\{\mathtt{int}, \mathtt{int} \to \mathtt{int}\}$ overloaded primitive fails);
- Concrete property dependent (e.g. restriction of polymorphic typings to certain environments only).
- The abstraction is rough and specific, not convenient for most program analyzes.

What is a Type System?

Lattice of Type Abstract Interpretations

- We can define a partial preorder on type systems through the notion of abstraction;
- In this way, type systems can be organized into a complete lattice;
- Type systems can then be defined as any abstraction of a type collecting semantics, the most refined of all of them;
- Type inference algorithms are the computable ones.

Church/Curry monotype semantics is an abstraction of Church/Curry polytype semantics

The correspondence is given by the Galois connection:

$$\langle \mathbb{T}^{\text{PC}},\, \supseteq\rangle \;\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}\; \langle \mathbb{T}^{\text{C}},\, \supseteq\rangle$$

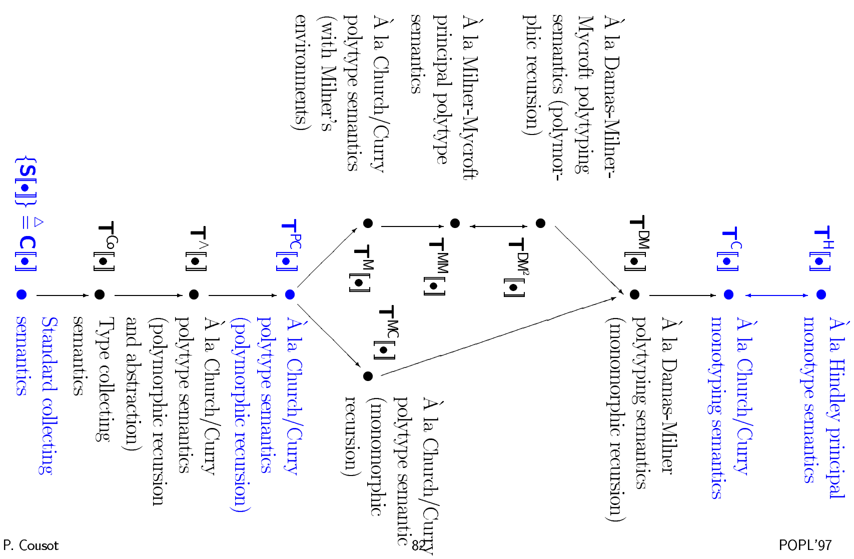
defined by:

$$\alpha(T) \stackrel{\Delta}{=} \{\langle H,\, m\rangle \mid \langle \Lambda \mathrm{y} \in \mathbb{X} \cdot \{H(\mathrm{y})\},\, m\rangle \in T\}$$

$$\gamma(T') \stackrel{\Delta}{=} \{\langle \Lambda \mathrm{y} \in \mathbb{X} \cdot \{H(\mathrm{y})\},\, m\rangle \mid \langle H,\, m\rangle \in T'\}$$

such that:

$$\mathbf{T}^{\text{C}}[\![e]\!] = \alpha(\mathbf{T}^{\text{PC}}[\![e]\!])$$



More in the Proceedings

- Sketch of a conjunctive Church/Curry polytype semantics (with polymorphic abstraction);
- More developments on abstraction and soundness;
- Design of a type collecting semantics;
- Proof that the Milner/Mycroft polymorphic type semantics is an Herbrand abstraction of the new Church/Curry polytype semantics;
- Proof that the Damas/Milner polymorphic type semantics is a further approximation of the Milner/Mycroft semantics;

Conclusion

A Personal Conclusion

- Positive:
  - Abstract interpretation provides a semantic foundation of type theory;
  - This leads to less empirical, calculation based development of type systems;
- Negative:
  - The Herbrand abstraction is too specific to be of general use in program analysis.

A Personal Hope

- Now that type inference is understood as an abstract interpretation it becomes possible to combine type inference with program analysis:
  - certainly not by using an Herbrand-like encoding of program properties;
  - probably in combination with other abstract domains.