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

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


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 polymtype system;
- Type inference and its limits in the context of program analysis.
- What is a type system?

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?

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.

Standard Semantics

- The standard semantics specifies the possible runtime behaviors of programs:
  \[ S \quad \text{semantic domain} \]
  \[ S[\bullet] \in E \rightarrow S \quad \text{standard semantics} \]

Syntax

- The syntax defines a set of valid programs:
  \[ e \in E \quad \text{programs/expressions} \]

Concrete Properties

- A concrete property of a program is a set of possible program behaviors;
- The set of concrete properties:
  \[ P \in P \triangleq \wp(S) \]
  is a complete boolean lattice:
  \[ \langle P, \subseteq, \emptyset, S, \cup, \cap, \neg \rangle \]
  for subset inclusion \( \subseteq \), that is logical implication.
Example of Concrete Property

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

$$\{AR^* \land n^* (n \in \mathbb{Z} \land 0 \leq n \land n! \leq \text{maxint} ? n! | \Omega), \land AR^* \land n^* (n \in \mathbb{Z} \land n < 0 ? \bot | n \in \mathbb{Z} \land 0 \leq n \land n! \leq \text{maxint} ? n! | \Omega)\}$$

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

$$\langle T, \leq, 0, 1, \lor, \land \rangle$$

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

Collecting Semantics

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

$$C[*] \in E \mapsto P$$

- The standard collecting semantics:

$$C[e] \triangleq \{S[e]\}$$

is the strongest concrete property.

Abstraction/Concretization

- The correspondence between concrete and abstract properties is defined by a Galois connection\(^1\):

$$\langle P, \subseteq, \emptyset, S, U, \cap \rangle \xrightarrow{\gamma} \langle T, \leq, 0, 1, \lor, \land \rangle$$

- $\alpha$: abstraction;
- $\gamma$: concretization.

Galois Connection

- By definition:
  \[
  \langle P, \subseteq, \emptyset, S, U, \cap \rangle \overset{\gamma}{\longrightarrow} \langle T, \leq, 0, 1, \lor, \land \rangle
  \]
  means:
  \[
  \forall P \in \mathcal{P} : \forall T \in \mathcal{T} : \alpha(P) \leq T \iff P \subseteq \gamma(T)
  \]
- The intuition is that:
  - \( \alpha(P) \) is the best/strongest/most precise approximation of \( P \);
  - \( \gamma(T) \) is the meaning of \( T \).

The Abstract Interpretation Design Methodology

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

Abstract Semantics

- An abstract semantics associates a abstract property to each program:
  \[
  T[e] \in E \leftrightarrow T
  \]
- The abstract semantics is a safe approximation of the collecting semantics:
  \[
  C[e] \subseteq \gamma(T[e])
  \]

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.

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

$$x, f, \ldots \in X : \quad \text{variables}$$

$$e \in E : \quad \text{expressions}$$

$$e ::= x \quad \text{variable}$$

$$\lambda x \cdot e \quad \text{abstraction}$$

$$e_1(e_2) \quad \text{application}$$

$$\mu f \cdot \lambda x \cdot e \quad \text{recursion}$$

$$1 \quad \text{one}$$

$$e_1 - e_2 \quad \text{difference}$$

$$(e_1 ? e_2 : e_3) \quad \text{conditional}$$
Semantic Domains

Ω wrong/runtime error value
⊥ non-termination
W \triangleq \{Ω\}
\_ \in \_ integers
\_ \in \_ values
R \in R \triangleq X \rightarrow \_ environments
\_ \in S \triangleq R \rightarrow \_ semantic domain

\footnotesize{\textsuperscript{2} [U \rightarrow U^*]: \text{continuous, left-adjunct, distinct functions from values' U to values' U.}}

Denotational Semantics of the Eager Lambda Calculus

- The denotational semantics is:
  \[ S[\_] \in E \rightarrow S \]

- The semantics S[1]R of constant 1 is the integer value 1:
  \[ S[1]R \triangleq 1 \]

- The semantics S[e_1 - e_2]R of a difference e_1 - e_2:
  \[
  S[e_1 - e_2]R \triangleq (S[e_1]R = \bot \lor S[e_2]R = \bot \lor \\
  \quad \quad S[e_1]R = z_1 \land S[e_2]R = z_2 \land z_1 - z_2 = \bot)
  \]

  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 \ ? \ e_2 : e_3) is a test for zero:
  \[
  S[(e_1 \ ? \ e_2 : e_3)]R \triangleq (S[e_1]R = \bot \lor \\
  \quad \quad S[e_1]R = 0 \lor S[e_2]R \\
  \quad \quad S[e_1]R = z \neq 0 \land S[e_3]R)
  \]

  - the evaluation does not terminate if e_2 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 $S[x]R$ of variable $x$ in environment $R$ is the value $R(x)$ of $x$ in $R$:

$$S[x]R \triangleq R(x)$$

• The semantics $S[e_1(e_2)]R$ of an application $e_1(e_2)$:

$$S[e_1(e_2)]R \triangleq (S[e_1]R = \bot \lor S[e_2]R = \bot \lor$$

$$\mid S[e_1]R = f \in [\mathbb{U} \mapsto \mathbb{U}] \land f(S[e_2]R)$$

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 $f = S[e_1]R$,
and the result is the application of $f$ to the value $S[e_2]R$ of $e_2$.

1. The evaluation goes wrong.
2. The result may be $\bot$ in case of non-termination of the call or if it goes wrong.

A recursive definition $\mu f \cdot \lambda x \cdot e$ defines a function $\varphi$ as the abstraction $\lambda x \cdot e$ where every occurrence of variable $x$ within the body $e$ refers to $\varphi$:

$$S[\mu f \cdot \lambda x \cdot e]R \triangleq \lambda p \in \equiv S[\lambda x \cdot e][f \leftarrow \varphi]$$

The choice of a least fixpoint for Scott-ordering $\equiv$ ensures that no result can be returned before the computation ends.
The let construct is defined such that:

\[ S[\text{let } x = e_1 \text{ in } e_2] \triangleq S[(\lambda x \cdot e_2)(e_1)] \]

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

Standard Collecting Semantics

- The standard collecting semantics:
  
  \[ C[\ast] \in E \mapsto P \]
  
  \[ C[e] \triangleq (S[e]) \]

  is the strongest concrete property.

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.

Church/Curry Monotypes

- Simple types are monomorphic:
  
  \[ m \in M^c, \quad m ::= \text{int} | m_1 \rightarrow m_2 \quad \text{monotype} \]

- A type environment associates a type to free program variables:
  
  \[ H \in H^c \triangleq X \mapsto M^c \quad \text{type environment} \]
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:
  \[ \theta \in \Gamma \triangleq \Gamma \times M \quad \text{typing} \]

• An abstract property or program type is a set of typings:
  \[ T \in \Gamma \triangleq \chi(\Gamma) \quad \text{program type} \]

Concretization Function

The meaning of types is a program property, as defined by the concretization function \( \gamma^c \):  

• Monotypes \( \gamma^c_\theta \in M^c \mapsto \varphi(\Upsilon) \):
  \[ \gamma^c_\theta(\text{int}) \triangleq \mathbb{Z} \cup \{\perp\} \]
  \[ \gamma^c_\theta(m_1 \rightarrow m_2) \triangleq \{ \varphi \in [\Upsilon \mapsto \Upsilon] \mid \forall u \in \gamma^c_\theta(m_2) : \varphi(u) \in \gamma^c_\theta(m_1) \} \cup \{\perp\} \]

  \(\text{for short, apply lifting/injection are omitted.}\)

• Type environment \( \gamma^c_\chi \in \Gamma^c \mapsto \varphi(\Upsilon) \):
  \[ \gamma^c_\chi(H) \triangleq \{ R \in \Upsilon \mid \forall x \in X : R(x) \in \gamma^c_\chi(H(x)) \} \]

• Typing \( \gamma^c_\theta \in \Gamma^c \mapsto \Upsilon^c \):
  \[ \gamma^c_\theta(\langle H, m \rangle) \triangleq \{ \phi \in \mathbb{S} \mid \forall R \in \gamma^c_\theta(H) : \phi(R) \in \gamma^c_\theta(m) \} \]

• Program type \( \gamma^c_\chi \in \Gamma^c \mapsto \Upsilon^c \):
  \[ \gamma^c_\chi(T) \triangleq \bigwedge_{\theta \in T} \gamma^c_\theta(\theta) \]
  \[ \gamma^c_\chi(\emptyset) \triangleq \mathbb{S} \]

• Types exclude going wrong:
  \[ \Omega \in \gamma^c_\chi(T) \iff T = \emptyset \]
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:

\[ \langle \mathbb{P}, \subseteq, \emptyset, \exists, \cap, \cup \rangle \xrightarrow{\alpha} \langle \mathbb{T}, \supseteq, \top, \emptyset, \forall, \cap, \cup \rangle \]

Church/Curry Monotyping Rules

\[ H \vdash^c x \Rightarrow H(x) \]  \hspace{1cm} (VAR)

\[ H \vdash^c e \Rightarrow m_1 \rightarrow m_2, \; H \vdash^c e_2 \Rightarrow m_1 \rightarrow m_2 \]

\[ H \vdash^c e_1 \cdot e_2 \Rightarrow m_2 \]  \hspace{1cm} (APP)

\[ H[x \leftarrow m_1] \vdash^c e \Rightarrow m_2 \]

\[ H \vdash^c \lambda x \cdot e \Rightarrow m_1 \rightarrow m_2 \]  \hspace{1cm} (ABS)

Abstract Semantics

- These typing rules can be understood as a compositional abstract semantics:
  \[ T[x] \in E \Rightarrow T' \]

- 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

**Church/Curry Monotype Semantics**

\[ T^\gamma[H] = \{ (H, H(x)) | H \in \mathbb{H}^\gamma \} \]  
\[ (\text{VAR}) \]

\[ T^\gamma[\lambda x \cdot e] = \{ (H, m_1 \rightarrow m_2) | \langle H[x\leftarrow m_1], m_2 \rangle \in T^\gamma[e] \} \]  
\[ (\text{ABS}) \]

\[ T^\gamma[e_1(e_2)] = \{ (H, m_2) | (H, m_1 \rightarrow m_2) \in T^\gamma[e_1] \land (H, m_1) \in T^\gamma[e_2] \} \]  
\[ (\text{APP}) \]

**Abstraction Theorem**

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

\[ \alpha^\circ(C[e]) \supseteq T^\gamma[e] \]
\[ \iff C[e] \subseteq \gamma^\circ(T^\gamma[e]) \]
\[ \iff S[e] \in \gamma^\circ(T^\gamma[e]) \]

\(^7\) Supper concrete and supper (i.e., collection) abstract approximation

**Typable Programs Cannot Go Wrong**

- A program \( e \) is typable iff \( T^\gamma[e] \neq \emptyset \).
- Typable programs cannot go wrong:

\[ (H, m) \in T^\gamma[e] \land R \in \gamma^\circ(H) \Rightarrow S[e]R \neq \Omega \]
**Design Methodology**

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

\[
C[e] \subseteq \gamma^c(T[e])
\]

\[
\iff \alpha^c(C[e]) \supseteq T[e]
\]

**provides a design methodology:**

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

---

**A Simplicitic Example ... Constants**

\[
T^c[1] \equiv \alpha^c(C[1])
\]

- best possible choice

\[
= \alpha^c(\{S[1]\})
\]

- def. collecting sem.

\[
= \alpha^c(\{\Lambda R \cdot 1\})
\]

- def. standard sem.

\[
= \bigcup \{T \mid \{\Lambda R \cdot 1\} \subseteq \gamma^c(T)\}
\]

- Galois connection

\[
= \bigcup \{T \mid \forall \theta \in T : \Lambda R \cdot 1 \in \gamma^c(\theta)\}
\]

- def. $\gamma^c$

\[
\{\theta \mid \Lambda R \cdot 1 \in \gamma^c(\theta)\}
\]

- set theory

\[
\{H, m\} \mid \forall R \in \gamma^c(H) : 1 \in \gamma^c(m)\}
\]

- def. $\gamma^c$

\[
\{H, \text{int}\} \mid H \in \mathbb{H}^c
\]

- def. $\gamma^1$
**Church/Curry Polytypes**

\[ m \in M^c, \quad m ::= \text{int} \mid m_1 \rightarrow m_2 \quad \text{monotype} \]

\[ p \in P^c \triangleq \wp(M^c) \quad \text{polytype} \]

\[ H \in H^c \triangleq X \rightarrow P^c \quad \text{type environment} \]

\[ \theta \in \Pi^c \triangleq H^c \times M^c \quad \text{typing} \]

\[ T \in T^c \triangleq \wp(P^c) \quad \text{program type} \]

Polymorphism is restricted to environments.

**Galois Connection**

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

\[ \langle P, \subseteq \rangle \overset{\gamma^c_1}{\longrightarrow} \langle T^c, \supseteq \rangle \]

- Concretization of monotypes \( \gamma^c_1 \in M^c \rightarrow \wp(\mathbb{U}) \) (unchanged):

\[ \gamma^c_1(\text{int}) \triangleq \mathbb{Z} \cup \{ \perp \} \]

\[ \gamma^c_1(m_1 \rightarrow m_2) \triangleq \{ \phi \in [\mathbb{U} \rightarrow \mathbb{U}] \mid \forall u \in \gamma^c_1(m_1) : \phi(u) \in \gamma^c_1(m_2) \} \cup \{ \perp \} \]

- Concretization of polytypes \( \gamma^c_1 \in M^c \rightarrow \wp(\mathbb{U}) \):

\[ \gamma^c_1(H, m) \triangleq \{ \phi \in \mathbb{S} \mid \forall R \in \gamma^c_1(H) : \phi(R) \in \gamma^c_1(m) \} \]

- Concretization of program types \( \gamma^c \in T^c \rightarrow \wp(\mathbb{U}) \):

\[ \gamma^c(T) \triangleq \bigcap_{\theta \in T} \gamma^c_1(\theta) \]

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

- If \( T \neq \emptyset \) then \( \gamma^c(T) \) excludes semantical values going wrong.
Church/Curry Polytype Semantics

- The Church/Curry polytype semantics:

\[ T^\kappa[\cdot] \in E \mapsto T^\kappa \]

is designed according to the soundness condition:

\[ \alpha^\kappa(C[\cdot]) \supseteq T^\kappa[\cdot] \]

\[ C[\cdot] \subseteq \gamma^\kappa(T^\kappa[\cdot]) \]

\[ S[\cdot] \subseteq \gamma^\kappa(T^\kappa[\cdot]) \]

so that typable programs cannot go wrong;

Church/Curry Polytyping Rules

- Rule-based presentation of the polytype semantics \( T^\kappa[\cdot] \):

\[
\frac{m \in H(x)}{H \vdash \kappa \; x \Rightarrow m} \quad \text{(VAR)}
\]

\[
\frac{H[x \leftarrow \{m_1\}] \vdash \kappa \; e \Rightarrow m_2}{H \vdash \kappa \; \lambda x \cdot e \Rightarrow m_1 \Rightarrow m_2} \quad \text{(APP)}
\]

- By calculation we derive a compositional functional fixpoint definition of \( T^\kappa[\cdot] \);
- We then express this polytype semantics in equivalent rule-based form.

\[
\frac{H \vdash \kappa \; e_1 \Rightarrow m_1 \Rightarrow m_2, \; H \vdash \kappa \; e_2 \Rightarrow m_1}{H \vdash \kappa \; e_1(e_2) \Rightarrow m_2} \quad \text{(ABS)}
\]

\[
\frac{P_1 \neq \emptyset, \; \forall m_1 \in P_1 : H \vdash \kappa \; e_1 \Rightarrow m_1, \; H[x \leftarrow P_1] \vdash \kappa \; e_2 \Rightarrow m_2}{H \vdash \kappa \; \text{let } x = e_1 \text{ in } e_2 \Rightarrow m_2} \quad \text{(LET)}
\]

\[
\forall m_1 \in P_1 : H[f \leftarrow P_1] \vdash \kappa \; \lambda x \cdot e \Rightarrow m_1, \; m \in P_1 \quad H \vdash \kappa \; \mu f \cdot \lambda x \cdot e \Rightarrow m \quad \text{(REC)}
\]
Church/Curry Polytyping Semantics

The most interesting case is for recursion:

\[ T^\tau[\mu x \cdot \lambda y \cdot e] \triangleq \{ \langle H, m \rangle \mid m \in g\mathbb{P} \subseteq M^\tau \Rightarrow M^\tau, \Psi \} \]

where \( \Psi \triangleq \wedge \mathbb{P}\{m' \mid \langle H[f \leftarrow p], m' \rangle \in T^\tau[\lambda x \cdot e] \} \)
or equivalently:

\[ T^\tau[\mu x \cdot \lambda y \cdot e] = \{ \langle H, m \rangle \mid \exists p \subseteq M^\tau \Rightarrow M^\tau : m \in p \wedge \forall m' \in p : \langle H[f \leftarrow p], m' \rangle \in T^\tau[\lambda x \cdot e] \} \]

\[ \text{P. Cousot} \quad 61 \quad \text{POPL'97} \]

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:

\[ F f g n x = g(f^n(x)) \]

has type:

\[ \{ \langle H, (m_1 \rightarrow m_1) \rightarrow (m_2 \rightarrow m_2) \rightarrow (\text{int} \rightarrow (m_1 \rightarrow m_2)) \rangle \mid H \in H^\tau \wedge m_1, m_2 \in M^\tau \} \]

\[ \text{P. Cousot} \quad 62 \quad \text{POPL'97} \]

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 properly and language-dependent abstractions which are not general enough for program analysis\(^9\).

\[ \text{P. Cousot} \quad 63 \quad \text{POPL'97} \]

\[ ^8 \text{convergence may have to be enforced through widening & narrowing} \]
\[ ^9 \text{Mostly applicable to boolean abstract domains} \]
Type Inference and its Limits for Program Analysis

Monotypes with Variables

\[ \forall a \in \mathcal{V} \]
\[ \tau \in M^{\eta} \]
\[ \tau ::= \text{int} \mid \forall a \mid \tau_1 \Rightarrow \tau_2 \]
\[ H \in H^{\eta} \triangleq X \mapsto M^{\eta} \]
\[ T \in T^{\eta} \triangleq H^{\eta} \times M^{\eta} \]

Herbrand Abstraction

- The Herbrand abstraction \( \text{log} \) (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 \( \forall a, \forall b, \ldots \) renaming:

\[ \text{log}(\{(m_1 \Rightarrow m_2) \Rightarrow ((m_1 \Rightarrow m_2) \Rightarrow (\text{int} \Rightarrow (m_1 \Rightarrow m_2))) \mid m_1, m_2 \in M^{\eta}\}) = (\forall a \Rightarrow \forall a) \Rightarrow ((\forall a \Rightarrow \forall b) \Rightarrow (\text{int} \Rightarrow (\forall a \Rightarrow \forall b))) \]

Concretization

\[ \text{ground}(\emptyset) = \emptyset \]
\[ \text{ground}(\text{int}) = \{\text{int}\} \]
\[ \text{ground}(\forall a \Rightarrow \forall a) = \{m \Rightarrow m \mid m \in M^{\eta}\} \]
\[ \text{ground}(\forall a) = M^{\eta} \]
Galois Connection

- The Herbrand abstraction \( \text{leg} \) is a Galois connection:

\[
\langle \varphi(\text{ground}(T)), \subseteq, \emptyset, \text{ground}(T), \cup, \cap \rangle \\
\xrightarrow{\text{ground}} \langle T^\varphi, \subseteq, \emptyset, [\alpha]^\varphi, \text{leg}, \text{gci} \rangle
\]

where:
- \( T \): set of terms with variables \( \alpha, \ldots \),
- \( \text{leg} \): least common generalization,
- \( \text{ground} \): set of ground instances,
- \( \subseteq \): instance preordering,
- \( \text{gci} \): greatest common instance.

Wrong Direction of Approximation

- The Church/Curry monotype abstraction:

\[
\langle \Pi, \subseteq, \emptyset, \exists, \cup, \cap \rangle \xleftarrow{\text{ground}} \langle T^\Pi, \subseteq, \emptyset, [\alpha]^\Pi, \text{leg}, \text{gci} \rangle
\]

- The Herbrand abstraction:

\[
\langle \Pi, \subseteq, \emptyset, \exists, \cup, \cap \rangle \xrightarrow{\text{ground}} \langle T^\Pi, \subseteq, \emptyset, [\alpha]^\Pi, \text{leg}, \text{gci} \rangle
\]

They do not compose!

Imprecision

- The Herbrand abstraction \( \text{leg} \) approximates a set of monotypes by a single monotype with variables;
- The Herbrand abstraction \( \text{leg} \) is quite approximate:

\[
\text{leg} \{ \text{int}, \text{int} \rightarrow \text{int} \} = \alpha
\]

- This e.g. excludes an overloaded primitive \( f \) which would behave has an \( \text{int} \rightarrow \text{int} \) function in a call context and as an \( \text{int} \) (e.g. \( f(\alpha) \)) in the context of an arithmetic operand.

A Specific Solution: Exact Herbrand Abstraction

- For soundness, we need a \( \subseteq \)-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:

\[ T^*[c] \in E \leftrightarrow \langle T^*, \subseteq \rangle \]

\[ \langle T^*, \subseteq \rangle \xrightarrow{\text{ground}} \langle T^* \rangle \varepsilon, \subseteq \]

We design, by calculation:

\[ T^*[c] = \text{lg}(T^*[c]) \]

and, for soundness, require:

\[ \text{ground}(T^*[c]) = T^*[c] \]

Design of the Type Semantics/Inference Algorithm

- Define \( T^*[c] \) by calculation of the expression \( \alpha''(T^*[c]) \) (\( \alpha'' \) based on \( \text{lg} \));
- Check that \( \gamma''(T^*[c]) = T^*[c] \) to ensure soundness (\( \gamma'' \) based on \( \text{ground} \));
- 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.

Exact Abstract Semantics

- It follows that Church/Curry monotype semantics \( T^*[c] \) and Hindley monotype with variables semantics \( T^*[c] \) are \( \langle \text{lg}, \text{ground} \rangle \) isomorphic;
- For other points of \( T^* \), outside \{ \( T^*[c] \mid c \in E \) \}, the approximation may be unsound;
- So the language \( E \) has to be somewhat singular, since the slightest change in the standard semantics might be unsound.

Hindley Monotype Semantics

\[ T''[x] \triangleq \langle H, H(x) \rangle \]

\[ T''[\lambda x \cdot e] \triangleq (T''[e] = \langle H, \tau \rangle \land \langle H[x \leftarrow \alpha], H(\tau) \rightarrow \tau \rangle \mid \emptyset) \]

\[ T''[e_1(e_2)] \triangleq (T''[e_2] = \langle H_2, \tau_2 \rangle \land g v \{ T''[e_1], \langle H_2, \tau_2 \rightarrow \alpha \rangle \} = \langle H, \tau \rangle \land \emptyset) \]

\[ T''[\mu x \cdot e] \triangleq (T''[\lambda x \cdot e] = \langle H, \tau \rangle \land \sigma = \text{mgv} \{ \alpha \rightarrow H(\tau), \tau \} \neq \emptyset \land \langle \sigma(H)[x \leftarrow \alpha], \sigma(\tau) \rangle \mid \emptyset) \]
\[ T^*[1] \triangleq \langle H, \text{int} \rangle \]
\[ T^*[e_1 \rightarrow e_2] \triangleq \text{gcd}\{\langle H, \text{int} \rangle, T^*[e_1], T^*[e_2]\} \]
\[ T^*[e_1 \ ? \ e_2 : e_3] \triangleq \begin{cases} T^*[e_1] = \langle H_1, \text{int} \rangle ? \\
\text{gcd}\{\langle H_1, \text{var} \rangle, T^*[e_2], T^*[e_3]\} | \emptyset \end{cases} \]

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

**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:

\[
\langle T, \cdot \rangle \xrightarrow{\alpha} \langle T', \cdot \rangle
\]

defined by:

\[
\alpha(T) \triangleq \{ \langle H, m \rangle \mid \langle \forall y \in X \cdot \{ H(y) \}, m \rangle \in T \}
\]

\[
\gamma(T') \triangleq \{ \langle \forall y \in X \cdot \{ H(y) \}, m \rangle \mid \langle H, m \rangle \in T' \}
\]

such that:

\[
T'[c] = \alpha(T'[c])
\]

---

**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.