Applicative Intersection Types

December 5, 2022

Xu Xue [1] Bruno C. d. S. Oliveira [1] Ningning Xie [2]

[1]University of Hong Kong [2]University of Cambridge

Intersection Types

- A term e having the type $A \& B$ means e has both A and B.

[1] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. “Functional characters of solvable terms”. In: *Mathematical Logic Quarterly* 27.2-6 (1981), pp. 45–58.

Intersection Types

- A term e having the type $A \,\&\, B$ means e has both A and B.
- Originally introduced by Coppo et al.[1], it allows $\lambda x.\, x\, x$ to be typed $((A \to B) \,\&\, A) \to B$.

[1] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. "Functional characters of solvable terms". In: *Mathematical Logic Quarterly* 27.2-6 (1981), pp. 45–58.

Intersection Types

- A term e having the type $A \& B$ means e has both A and B.
- Originally introduced by Coppo et al.[1], it allows $\lambda x.\, x\, x$ to be typed $((A \to B) \& A) \to B$.
- In languages like TypeScript, the intersection types are explicitly inhabitated.

```
interface Name { name: string; }
interface ID { id: number; }
type Person = Name & ID
let e : Person = { id: 42, name: 'Alice'};
```

[1] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. "Functional characters of solvable terms". In: *Mathematical Logic Quarterly* 27.2-6 (1981), pp. 45–58.

Merge Operator[3]

- $e_1, , e_2$ means it can be used as e_1 or e_2.

[2] We use bidirectional typing, $\Gamma \vdash e \Leftrightarrow A$, and $\Leftrightarrow ::= \Leftarrow | \Rightarrow$

[3] Jana Dunfield. “Elaborating intersection and union types”. In: *Journal of Functional Programming* 24.2-3 (2014), pp. 133–165.

Merge Operator[3]

- $e_1 ,, e_2$ means it can be used as e_1 or e_2.
- Force intersection types to be *explicitly* introduced and inhabitated.
- Typing for merge is [2]

$$\text{T-MRG}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A \,\&\, B}$$

[2] We use bidirectional typing, $\Gamma \vdash e \Leftrightarrow A$, and $\Leftrightarrow ::= \Leftarrow | \Rightarrow$

[3] Jana Dunfield. "Elaborating intersection and union types". In: *Journal of Functional Programming* 24.2-3 (2014), pp. 133–165.

Merge Operator[3]

- $e_1 ,, e_2$ means it can be used as e_1 or e_2.
- Force intersection types to be *explicitly* introduced and inhabitated.
- Typing for merge is [2]

$$\textsc{T-Mrg}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A \,\&\, B}$$

- Merge operator adds expressive power and enables many applications.

[2]We use bidirectional typing, $\Gamma \vdash e \Leftrightarrow A$, and $\Leftrightarrow ::= \Leftarrow | \Rightarrow$

[3]Jana Dunfield. "Elaborating intersection and union types". In: *Journal of Functional Programming* 24.2-3 (2014), pp. 133–165.

Extensible Records[4]

- Records can be represented by *syntactic sugar of merge operator*.
- $\{x = e_1, y = e_2, z = e_3\}$ can be viewed as $\{x = e_1\}, , \{y = e_2\}, , \{z = e_3\}$.

[4] Luca Cardelli and John C Mitchell. "Operations on records". In: *Mathematical structures in computer science* 1.1 (1991), pp. 3–48.

Extensible Records[4]

- Records can be represented by *syntactic sugar of merge operator*.
- $\{x = e_1, y = e_2, z = e_3\}$ can be viewed as $\{x = e_1\},,\{y = e_2\},,\{z = e_3\}$.
- Record width subtyping *for free*.

$$\{l_i : T_i\}^{i=1..n..n+k} <: \{l_i : T_i\}^{1..n}$$

is subsumed by

$$\{l_1 : A\} \,\&\, \{l_2 : B\} <: \{l_1 : A\}$$

is subsumed by

$$A \,\&\, B <: A$$

[4] Luca Cardelli and John C Mitchell. “Operations on records”. In: *Mathematical structures in computer science* 1.1 (1991), pp. 3–48.

Record Projection

- Record Projection is standard.

$$(\{x = e_1\} ,, \{y = e_2\}).x \hookrightarrow e_1$$

$$(\{x = e_1\} ,, \{y = e_2\}).y \hookrightarrow e_2$$

- Record Concatenation is simply merging.

$$(\{x = e_1\} ,, \{y = e_2\}) ,, \{z = e_3\}$$

Overloaded Functions[5]

- Function implementation *varies* depending on the types of arguments.

[5] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. "A calculus for overloaded functions with subtyping". In: *Information and Computation* 117.1 (1995), pp. 115–135.

Overloaded Functions[5]

- Function implementation *varies* depending on the types of arguments.
- Consider Haskell's `show` function.

```
show :: Show a => a -> String
instance Show Int where
  show = showInt
instance Show Bool where
  show = showBool
-- instance will be selected according to the argument type
show 1 ↪ showInt 1 ↪ "1"
show true ↪ showBool true ↪ "true"
```

- `show` can be defined as `showInt,,showBool`

[5] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. "A calculus for overloaded functions with subtyping". In: *Information and Computation* 117.1 (1995), pp. 115–135.

Overloaded Application

- Overloaded Application is standard.

```
show : (Int -> String) & (Bool -> String)
show = showInt,,showBool
show 1 ↪ showInt 1 ↪ "1"
show true ↪ showBool true ↪ "true"
```

- Adding overloading instances is simply by merging.

```
newShow = show,,showDouble
```

Return type Overloading[6]

- Function implementation varies depending on the surrounding contexts.

[6] Koar Marntirosian et al. "Resolution as Intersection Subtyping via Modus Ponens". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020).

Return type Overloading[6]

- Function implementation varies depending on the surrounding contexts.
- Consider Haskell's read function

```
read :: Read a => String -> a
instance Read Int where
  read = readInt
instance Read Bool where
  read = readBool
-- instance will be selected according to surrounding contexts
succ (read "1") ↪ 2
not (read "true") ↪ false
```

[6] Koar Marntirosian et al. "Resolution as Intersection Subtyping via Modus Ponens". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020).

Return type Overloading[6]

- Function implementation varies depending on the surrounding contexts.
- Consider Haskell's `read` function

```
read :: Read a => String -> a
instance Read Int where
  read = readInt
instance Read Bool where
  read = readBool
-- instance will be selected according to surrounding contexts
succ (read "1") ↪ 2
not (read "true") ↪ false
```

- Calculi with merge operator can do in a similar way.

```
read : (String -> Int) & (String -> Bool)
read = readInt,,readBool
```

[6] Koar Marntirosian et al. "Resolution as Intersection Subtyping via Modus Ponens". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020).

Nested Composition[7]

- It reflects *distributivity* of intersection types at the term level.

$$\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \quad \text{S-Distri-Rcd}$$

$$(A \rightarrow B) \,\&\, (A \rightarrow C) <: A \rightarrow (B \,\&\, C) \quad \text{S-Distri-Arr}$$

- Results extracted from nested terms will be composed when eliminating terms created by the merge operator.

[7] Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. "The essence of nested composition". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

Nested Composition via Projection and Application

- For records

$$(\{x = e_1\},, \{x = e_2\}).x \hookrightarrow e_1,, e_2$$

Nested Composition via Projection and Application

- For records

$$(\{x = e_1\} ,, \{x = e_2\}).x \hookrightarrow e_1 ,, e_2$$

- For overloaded functions

$$f : Int \rightarrow Int \rightarrow Int$$
$$g : Int \rightarrow Bool \rightarrow Bool$$
$$(f ,, g)\ 1 \hookrightarrow (f\ 1) ,, (g\ 1)$$

Nested Composition via Projection and Application

- For records

$$(\{x = e_1\},,\{x = e_2\}).x \hookrightarrow e_1,,e_2$$

- For overloaded functions

$$f : Int \to Int \to Int$$
$$g : Int \to Bool \to Bool$$
$$(f,,g)\;1 \hookrightarrow (f\;1),,(g\;1)$$

- Both cases are "unnatural"
  since we allow repeated labels and ambiguous overloaded application.

Goodness of Nested Composition

- *[Nested record composition]* Key feature of *Compositional Programming*[8].
  - solves the Expression Problem naturally.
  - models forms of family polymorphism.

[8] Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. "Compositional Programming". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43.3 (2021), pp. 1–61.

Goodness of Nested Composition

- *[Nested record composition]* Key feature of *Compositional Programming*[8].
  - solves the Expression Problem naturally.
  - models forms of family polymorphism.
- *[Nested function composition]* It enables *first-class curried overloaded functions*.
  - overloaded functions are default curried;
  - we can abstract and return overloaded functions in a flexible way;
  - it's a novel and interesting finding in this work.

[8] Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. "Compositional Programming". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43.3 (2021), pp. 1–61.

Challenges in Type Inference

In traditional calculi, we have the typing rule for application:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B} \ \textsc{T-App}$$

This does not apply to case `show 1`, where

$$\frac{\Gamma \vdash \mathit{show} \Rightarrow A \,\&\, B \qquad \Gamma \vdash e_2 \Leftarrow\ ?}{\Gamma \vdash e_1\ e_2 \Rightarrow\ ?} \ \textsc{T-App}$$

Challenges in Type Inference

A direct method is to:

1. assume we have the argument type A;
2. assume the type of function to be a intersection of function types:

$$(A_1 \to B_1) \,\&\, (A_2 \to B_2) \,\&\, \ldots \,\&\, (A_n \to B_n)$$

Challenges in Type Inference

A direct method is to:

1. assume we have the argument type A;
2. assume the type of function to be a intersection of function types:

$$(A_1 \rightarrow B_1) \,\&\, (A_2 \rightarrow B_2) \,\&\, \dots \,\&\, (A_n \rightarrow B_n)$$

3. then iterate intersection types by comparing the argument type A and input type A_i;

Challenges in Type Inference

A direct method is to:

1. assume we have the argument type A;
2. assume the type of function to be a intersection of function types:

$$(A_1 \rightarrow B_1) \,\&\, (A_2 \rightarrow B_2) \,\&\, \ldots \,\&\, (A_n \rightarrow B_n)$$

3. then iterate intersection types by comparing the argument type A and input type A_i;
4. compose the outputs as the result type

Challenges in Dynamic Semantics

A direct method is to:

1. assume the overloaded function to be a merge of functions,

Challenges in Dynamic Semantics

A direct method is to:

1. assume the overloaded function to be a merge of functions,
2. then select correct instances according to the types.

Challenges in Dynamic Semantics

A direct method is to:

1. assume the overloaded function to be a merge of functions,
2. then select correct instances according to the types.
    - call-by-value strategy
    - type-dependent semantics

Distributivity Breaks the Assumptions

```
pshow : Unit -> (Int -> String) & (Bool -> String)
pshow = λx. show
pshow unit 1 ↪ "1"
pshow unit true ↪ "true"
```

Distributivity Breaks the Assumptions

```
pshow : Unit -> (Int -> String) & (Bool -> String)
pshow = λx. show
pshow unit 1 ↪ "1"
pshow unit true ↪ "true"
```

- pshow is **not** a merge of functions (wrapped in a lambda);
- its type is **not** a intersection of function types;
- it's still treated as an overloaded function.

Re-interpret Subtyping

We can have two interpretations of $A <: B \to C$:

- Suppose A, B and C are given, we tell whether the subtyping holds.

$$(Int \to String) \,\&\, (Bool \to String) <: Int \to String$$

- Suppose A and B are given, we infer the result type C[9].

$$(Int \to String) \,\&\, (Bool \to String) <: Int \to \,?$$

[9] which is also the type of overloaded application.

Applicative Subtyping

$A \ll S$ is a specialized subtyping used to infer the type of applications and projections [10].

$$
\begin{aligned}
A_1 \to A_2 \ll B &= A_2 && \textit{when } B <: A_1 && (1)\\
A_1 \to A_2 \ll B &= . && \textit{when } \neg(B <: A_1) && (2)\\
\{l = A\} \ll l &= A && && (3)\\
\{l_1 = A\} \ll l_2 &= . && \textit{when } l_1 \neq l_2 && (4)\\
A_1 \& A_2 \ll S &= (A_1 \ll S) \circledcirc (A_2 \ll S) && && (5)\\
A \ll S &= . && \textit{otherwise} && (6)
\end{aligned}
$$

[10] $S ::= A \mid l$, Selector S is either type A or label l

Examples of Applicative Subtyping

```
show 1
```

$$
\begin{aligned}
& (Int \to String) \,\&\, (Bool \to String) \ll Int \\
\text{by } (5) \hookrightarrow\ & (Int \to String) \ll Int \circledcirc (Bool \to String) \ll Int \\
\text{by } (1)\ (2) \hookrightarrow\ & String \circledcirc .
\end{aligned}
$$

```
read "1"
```

$$
\begin{aligned}
& (String \to Int) \,\&\, (String \to Bool) \ll String \\
\text{by } (5) \hookrightarrow\ & (String \to Int) \ll String \circledcirc (String \to Bool) \ll String \\
\text{by } (1) \hookrightarrow\ & Int \circledcirc Bool
\end{aligned}
$$

Composition Operators

One version that implements *nested composition semantics* [11].

$$
\begin{aligned}
. \circledcirc . &= . \\
A_1 \circledcirc . &= A_1 \\
. \circledcirc A_2 &= A_2 \\
A_1 \circledcirc A_2 &= A_1 \,\&\, A_2
\end{aligned}
$$

[11] We have another version of the operator which models the overloading semantics

Examples (applying nested composition semantics)

$$\begin{aligned}
(Int \to String) \,\&\, (Bool \to String) &\ll Int &= String \\
(String \to Int) \,\&\, (String \to Bool) &\ll String &= Int \,\&\, Bool \\
\{x : String\} \,\&\, \{y : String\} &\ll y &= String
\end{aligned}$$

Let arguments go "together"

We infer both the type of function (merges) and argument together and then compute.

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A \ll B = C}{\Gamma \vdash e_1\, e_2 \Rightarrow C}\ \text{T-App}$$

Examples (applying nested composition semantics)

We assume Γ is $f : I \to I \to I, g : I \to B \to B$. [12]

$$\dfrac{\dfrac{\Gamma \vdash (f\,,, g) \Rightarrow (I \to I \to I)\ \&\ (I \to B \to B) \qquad \Gamma \vdash 2 \Rightarrow I}{\Gamma \vdash (f\,,, g)\ 2 \Rightarrow \boxed{(I \to I)\ \&\ (B \to B)}}\ \text{T-APP} \qquad \Gamma \vdash \mathit{true} \Rightarrow B}{\Gamma \vdash (f\,,, g)\ 2\ \mathit{true} \Rightarrow \boxed{B}}\ \text{T-APP}$$

1. $f\,,, g$
2. $(f\,,, g)\ 2$
3. $(f\,,, g)\ 2\ \mathit{true}$

[12] I stands for *Int*, B stands for *Bool*.

Metatheory

$$(Int \to String) \,\&\, (Bool \to String) \ll Int = String$$
$$(String \to Int) \,\&\, (String \to Bool) \ll String = Int \,\&\, Bool$$
$$\{x : String\} \,\&\, \{y : String\} \ll y = String$$

$$(Int \to String) \,\&\, (Bool \to String) <: Int \to String$$
$$(String \to Int) \,\&\, (String \to Bool) <: String \to Int \,\&\, Bool$$
$$\{x : String\} \,\&\, \{y : String\} <: \{y : String\}$$

Metatheory

Lemma (Soundness (Function))

If $A \ll B = C$, *then* $A <: B \to C$.

Lemma (Completeness (Function))

If $A <: B \to C$, *then* $\exists D, A \ll B = D \wedge D <: C$.

Calculi Syntax

Expressions	$e ::= x \mid i \mid e : A \mid e_1\, e_2 \mid \lambda x\,.e : A \to B \mid e_1 ,, e_2 \mid \{l = e\} \mid e.l$
Raw Values	$p ::= i \mid \lambda x\,.e : A \to B$
Values	$v ::= p : A^o \mid v_1 ,, v_2 \mid \{l = v\}$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$

- Values carry extra annotations as runtime types;
- The dispatching is based on runtime types;
- The restriction on runtime types settles a canonical form of overloaded functions.

Operational Semantics

Step-App

$$\frac{(v_1 \bullet v_2) \hookrightarrow e}{v_1\, v_2 \longmapsto e}$$

Step-Prj

$$\frac{(v \bullet l) \hookrightarrow v'}{v.l \longmapsto v'}$$

Applicative Dispatching [13]

$\boxed{(v \bullet vl) \ \hookrightarrow \ e}$ *(Applicative Dispatching)*

App-Lam

$$\frac{v \longmapsto_A v'}{((\lambda x.\, e : A \to B) : C \to D \bullet v) \ \hookrightarrow \ e[x \mapsto v'] : D}$$

App-Proj

$$\frac{}{(\{l = v\} \bullet l) \ \hookrightarrow \ v}$$

App-Mrg-L

$$\frac{\langle v_2 \rangle \ll \langle vl \rangle = . \qquad (v_1 \bullet vl) \ \hookrightarrow \ e}{((v_1 ,, v_2) \bullet vl) \ \hookrightarrow \ e}$$

App-Mrg-R

$$\frac{\langle v_1 \rangle \ll \langle vl \rangle = . \qquad (v_2 \bullet vl) \ \hookrightarrow \ e}{((v_1 ,, v_2) \bullet vl) \ \hookrightarrow \ e}$$

App-Mrg-P

$$\frac{\langle v_1 \rangle \ll \langle vl \rangle \neq . \qquad \langle v_2 \rangle \ll \langle vl \rangle \neq . \qquad (v_1 \bullet vl) \ \hookrightarrow \ e_1 \qquad (v_2 \bullet vl) \ \hookrightarrow \ e_2}{((v_1 ,, v_2) \bullet vl) \ \hookrightarrow \ e_1 ,, e_2}$$

[13] $\langle v \rangle$ extracts the runtime type of v

Type Soundness

Theorem (Preservation)

If $\cdot \vdash e \Leftrightarrow A$ *and* $e \longmapsto e'$, *then* $\cdot \vdash e' \Leftarrow A$.

Theorem (Progress)

If $\cdot \vdash e \Leftrightarrow A$, *then* e *is a value or* $\exists e', e \longmapsto e'$.

More in the paper

- Sound/complete lemmas in the settings of records.
- Three variants of sound/complete lemmas with regard to different subtyping.
- Second calculus with disjoint restriction, is proved to be type sound and deterministic.
- Racket interpreter implementation of the calculi.

Coq Formalisation & Interpreter Implementation
“https://github.com/juniorxxue/applicative-intersection"

Q & A.