Intersection Type with Determinism

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

1. Introduction

The benefit of a merge, compared to a pair, is that you don't need to explicitly extract an item out. For example, `fst (1,'c')`

Definition 1. Determinism If $e : \tau_1 \hookrightarrow E_1$ and $e : \tau_2 \hookrightarrow E_2$, then $\tau_1 = \tau_2$ and $E_1 = E_2$.

Coherence is a property about the relation between syntax and semantics. We say a semantics is *coherent* if the syntax of a term uniquely determines its semantics.

Definition 2. Coherence If $e_1 : \tau_1 \hookrightarrow E_1$ and $e_2 : \tau_2 \hookrightarrow E_2$, $E_1 \Downarrow v_1$ and $E_2 \Downarrow v_2$, then $v_1 = v_2$.

2. Theory

Types include three constructs: type variables, function types, and family intersections:

$$A, B, C ::= \alpha \mid A \to B \mid \cap_{\alpha \in S} A$$

A *family intersection*, denoted by $\cap_{\alpha \in S} A$, is the intersection of the family of types A indexed by α (So α may be free in A).

Let $\mathbb{U}$ be the universe of all types.

Then the parametric polymorphism is defined as:

$$\forall \alpha.\, A ::= \cap_{\alpha \in \mathbb{U}} A$$

Binary intersection is a special case of family intersection:

$$A \cap B ::= \cap_{\alpha \in \{A,B\}} \alpha$$

$$\frac{(1,,'c') : \mathsf{Int} \cap \mathsf{Char}}{(1,,'c'),,(2,,\mathsf{True})}$$

Definition 3. Disjointness Two types A and B are *disjoint* (written as "A, B disjoint") if there does not exist a type C such that $C <: A$ and $C <: B$ and $C <: A \cap B$.

$$\frac{A \text{ type} \qquad B \text{ type}}{A \cap B \text{ type}}\ \textsc{Formation} \qquad \frac{e : A \qquad e : B}{e : A \cap B}\ \textsc{Intro}$$

$$\frac{e : A \cap B}{e : A}\ \textsc{Elim}_1 \qquad \frac{e : A \cap B}{e : B}\ \textsc{Elim}_2$$

$$\frac{e_1 : A \qquad e_2 : B}{e_1,,e_2 : A}\ \textsc{Merge}_1 \qquad \frac{e_1 : A \qquad e_2 : B}{e_1,,e_2 : B}\ \textsc{Merge}_2$$

A well-formed type is such that given any query type, it is always clear which subpart the query is referring to.

2.1 Equational reasoning

We can define a `fst` function that extracts the first item of a merged value:

```
let fst A B (x : A & B) = (\(y : A). y) x in ...
```

Then we have the following equational reasoning:

```
fst Int Int (2,,3)
(\(y : Int). y) (2,,3)
```

2.2 Discussion

In our type-directed translation, some inference rules return conclusions having *the same constructor*. This phenomenon makes the translation nondeterministic. As an example,

```
({x=1},,{x=2}).x
```

can evaluate to either `1` or `2` (according their translation in the target language). In this case, the constructor is the intersection operator, for which both rules, (select1) and (select2), are applicable.

One remedy, which you may have realised, is to enforce the order of applying rules. Whenever the case as shown above happens, the right component of `&` and `,,` will take precedence. In other words, the (select2) rule is tried first. Only if (select2) fails, the (select1) rule is tried. Therefore, `({x=1},,{x=2}).x` can only evaluate to 2. Likewise, `({x=1},,{x="hi"}).x` will evaluate to `"hi"` and will be of type `String`. Generally, three pairs of rules in our system that cause nondeterminism can all be implemented in the same fashion (sub-and2 is favored over sub-and1), and (restrict2 is favored over restrict1).

This approach seem works fine until you think about how it interact with parametric polymorphism.

```
(/\A. \(x:A&Int). x) Int (1,,2) + 1
```

If we would like to have a deterministic elaboration result, another idea is to tweak the rules a little bit so that given a term, it is no longer possible that both of the twin rules described above can be used. For example, if $\tau_1 \cap \tau_2 <: \tau_3$, we would like to be certain that either $\tau_1 <: \tau_3$ holds or $\tau_2 <: \tau_3$ holds, but not both.

Formally, we can state this theorem as:

Theorem 1. *If* τ_1, τ_2, τ_3, *and* $\tau_1 \cap \tau_2$ *are well-formed types, and* $\tau_1 \cap \tau_2 <: \tau_3$, *then* $\tau_1 \cap \tau_3$ exclusive *or* $\tau_2 \cap \tau_3$.

Note that A *exclusive* or B is true if and only if their truth value differ. Next, we are going to investigate the minimal requirement (necessary and sufficient conditions) such that the theorem holds.

If τ_1 and τ_2 in this setting are the same, for example, $\mathsf{Int} \cap \mathsf{Int} <: \mathsf{Int}$, obviously the theorem will not hold since both the left Int and the right Int are a subtype of Int.

If our types include primitive subtyping such as $\mathsf{Nat} <:_{\text{prim}} \mathsf{Int}$ (a natural number is also an integer), which can be promoted to the normal subtyping with this rule:

$$\frac{\tau_1 <:_{\text{prim}} \tau_2}{\tau_1 <: \tau_2}$$

the theorem will also not hold because $\mathsf{Int} \cap \mathsf{Nat} <: \mathsf{Int}$ and yet $\mathsf{Int} <: \mathsf{Int}$ and $\mathsf{Nat} <: \mathsf{Int}$.

We can try to rule out such possibilities by making the requirement of well-formedness stronger. This suggests that the two types on the sides of $\cap$ should not "overlap". In other words, they should be "disjoint". It is easy to determine if two base types are disjoint. For example, Int and Int are not disjoint. Neither do Int and Nat. Also, types built with different constructors are disjoint. For example, Int and $\mathsf{Int} \to \mathsf{Int}$. For function types, disjointness is harder to visualise. But bear in the mind that disjointness can defined by the very requirement that the theorem holds.

We shall give two semantics and show the two are the same.

- an type-directed semantics
- a direct operational semantics

say the example above:

without the cast, you could either get: 1„'c' or 1 depending on what rules you use

but I think with your change, you can only get the first

(which is what we want)

let me see how we can get '1' before the change

With the change, we need $\mathsf{Int} <: \mathsf{Int} \cap \mathsf{Char}$ to hold in order to get the premise, which does not. So it can be shown that $(\mathsf{Int} \cap \mathsf{Char})((1, ,' c') : \mathsf{Int} \cap \mathsf{Char}) \hookrightarrow 1$ is not derivable.