Coherence

1. A Section

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.

> **Not convinced by the example.**

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

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 $\tau_1\&\tau_2 <: \tau_3$, 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 $\tau_1\&\tau_2$ are well-formed types, and $\tau_1\&\tau_2 <: \tau_3$, then $\tau_1\&\tau_3$ exclusive or $\tau_2\&\tau_3$.*

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, $Int\&Int <: Int$, 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 $Nat <:_{prim} Int$ (a natural number is also an integer), which can be promoted to the normal subtyping with this rule:

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

the theorem will also not hold because $Int\&Nat <: Int$ and yet $Int <: Int$ and $Nat <: Int$.

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 & 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 $Int \rightarrow Int$. 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 $Int <: Int\&Char$ to hold in order to get the premise, which does not. So it can be shown that $(Int\&Char)((1,,'c') : Int\&Char) \hookrightarrow 1$ is not derivable.