Distributive Iso-Recursive Subtyping

ANONYMOUS AUTHOR(S)

1 Introduction

2 Overview

2.1 Syntax

$$
\begin{array}{llcl}
\text{Source Types} & A, B & ::= & \mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid A_1 \& A_2 \mid \mu\alpha.\, A \mid A^{\alpha} \\
\text{Target Types} & A, B & ::= & \mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid A_1 \& A_2 \mid \zeta(\alpha, l).\, A \mid \{\alpha.\, A\}^{l}
\end{array}
$$

We will abuse the meta-variable $A, B, C, D, \ldots$ to denote both source and target types. We will indicate in the rule whether the type in question is a source type or a target type.

Note, though written as named representation, we allow the use of alpha renaming at will in all the rules. Specifically, if we take a locally nameless view, then for following type constructs:

- In source type, variables are bound to recursive types. In $\mu\alpha.A$, the variable α is bound to the type A.
- In target type, variables are bound to both recursive shell types and labeled types. In $\zeta(\alpha, l).A$ and $\{\alpha.A\}^l$, the variable α is bound to the type A.
  For example, in the type $\zeta(\alpha, l).\alpha \to \{\alpha.(\mathsf{nat} \to \alpha)\}^l$, the first α is bound to the recursive shell type and the α in $(\mathsf{nat} \to \alpha)$ is bound to the labeled type.
- In target type, labels can also be renamed up to α-equivalence w.r.t. the recursive shell type, i.e., in $\zeta(\alpha, l).A$, the label l is bound to the type A.

2.2 Translation

$\boxed{A \rightsquigarrow B}$ *(Source type A translates to target type B)*

$$
\textsc{trans-top}\ \frac{}{\top \rightsquigarrow \top}
\qquad
\textsc{trans-nat}\ \frac{}{\mathsf{nat} \rightsquigarrow \mathsf{nat}}
\qquad
\textsc{trans-var}\ \frac{}{\alpha \rightsquigarrow \alpha}
\qquad
\textsc{trans-and}\ \frac{A' \rightsquigarrow A \qquad B' \rightsquigarrow B}{A' \& B' \rightsquigarrow A \& B}
\qquad
\textsc{trans-arr}\ \frac{A' \rightsquigarrow A \qquad B' \rightsquigarrow B}{A' \to B' \rightsquigarrow A \to B}
$$

$$
\textsc{trans-mu}\ \frac{A' \rightsquigarrow A}{\mu\alpha.\, A' \rightsquigarrow \zeta(\alpha, l).\, (A[\alpha^- \mapsto \{\alpha.\, A\}^l])}
\qquad
\textsc{trans-label}\ \frac{A' \rightsquigarrow A}{A'^{\alpha} \rightsquigarrow \{\_.\, A\}^{\alpha}}
$$

Fig. 1. Translation rules.

In rule TRANS-MU, we perform a *bottom-up* translation, which means the body is first translated to a target type A, and then the translation result is a polarized substitution of the variable α in the type A.

The substitution type, $\{\alpha.A\}^l$, is a labeled type, and α is bound in the type A, so if $\mu\alpha.A'$ is closed, then $\{\alpha.A\}^l$ is also closed.

The substitution result is a shell type, the ζ (looking like a shell) is used to indicate the range of the original recursive type in the translation. The variable α is bound to the type $A[\alpha^- \mapsto \{\alpha.A\}^l]$.

2025. ACM XXXX-XXXX/2025/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

More precisely, only the free variable α in the first A will be bound to the shell type, as the α's in $\{\alpha.A\}^l$ will be shadowed by the binder in labeled type.

Meanwhile, the label l is bound to the shell type, indicated by the binder $\zeta(\alpha, l).A$ in the translation introducing two binders at the same time. The binded label l is used in the labeled type $\{\alpha.A\}^l$ to achieve the same effect of *nominal unfolding*. But unlike previous work which directly assigns fresh labels, we introduce labels as binded structures to ensure that two types can be *independently translated* and then compared in a subtyping relation.

2.3 Algorithmic Subtyping

The subtyping algorithm can be seen as a simple extension of Huang et al. [2021]'s BCD subtyping algorithm. The shell types are splittable (distributive over intersection) and have standard subtyping rule SUB-SHELL, without the need of extra unfolding as seen in the nominal unfolding rules. (Note, in the named representation, the body of the shell type can be compared directly, while with a locally nameless view, both α and l have to be opened to compare the body of the shell type.)

For the labeled types we also have the standard subtyping rule SUB-LABEL. However, since labeled types serve as simulation of double / nominal unfolding, they are not splittable.

$\boxed{A \leq B}$ *(Subtyping for the target types)*

$$\text{SUB-NAT}\ \frac{}{\mathsf{nat} \leq \mathsf{nat}} \qquad \text{SUB-TOP}\ \frac{}{A \leq \top} \qquad \text{SUB-VAR}\ \frac{}{\alpha \leq \alpha} \qquad \text{SUB-ARR}\ \frac{A_2 \leq A_1 \qquad B_1 \leq B_2}{A_1 \to B_1 \leq A_2 \to B_2} \qquad \text{SUB-SHELL}\ \frac{A \leq B}{\zeta(\alpha, l).\, A \leq \zeta(\alpha, l).\, B}$$

$$\text{SUB-LABEL}\ \frac{A \leq B}{\{\alpha.\, A\}^l \leq \{\alpha.\, B\}^l} \qquad \text{SUB-ANDL}\ \frac{A_1 \leq B}{A_1 \,\&\, A_2 \leq B} \qquad \text{SUB-ANDR}\ \frac{A_2 \leq B}{A_1 \,\&\, A_2 \leq B} \qquad \text{SUB-AND}\ \frac{A \leq B_1 \qquad A \leq B_2 \qquad B_1 \lhd B \rhd B_2}{A \leq B}$$

$\boxed{B_1 \lhd B \rhd B_2}$ *(Splitting target types)*

$$\text{SPL-AND}\ \frac{}{A \lhd A \,\&\, B \rhd B} \qquad \text{SPL-ARR}\ \frac{B_1 \lhd B \rhd B_2}{A \to B_1 \lhd A \to B \rhd A \to B_2} \qquad \text{SPL-SHELL}\ \frac{B_1 \lhd B \rhd B_2}{\zeta(\alpha, l).\, B_1 \lhd \zeta(\alpha, l).\, B \rhd \zeta(\alpha, l).\, B_2}$$

Fig. 2. Algorithmic subtyping rules.

With the algorithmic subtyping rules defined for translated types, we obtain an algorithm for the source types by first translating the source types to target types, and then applying the algorithmic subtyping rules to the translated types. (Note, in this document we typically use <: for subtyping relations on the source types and ≤ for the target types.)

$$A <:_a B \triangleq \forall A'\, B',\ \text{if } A \rightsquigarrow A' \wedge B \rightsquigarrow B' \text{ then } A' \leq B'$$

2.4 Declarative Subtyping

We wish to argue the correctness of the algorithmic subtyping by proving its soundness and completeness to declarative subtyping rules. The declarative rules in Figure 3 are basically the original BCD rules extended with

(1) The nominal unfolding rule for subtyping iso-recursive types. (rule SUB-MU)
(2) A (hypothetical) distributive rule for merging two recursive types. (rule SUB-DIST-MU)

(3) The toplike rule for recursive types. (rule Sub-top-mu)

Note that the declarative rule includes a built-in transitivity rule Sub-trans, which makes the rules non-algorithmic.

$\boxed{A <: B}$ *(Sub)*

$$\textsc{Sub-refl}\ \frac{}{A <: A} \qquad \textsc{Sub-trans}\ \frac{A <: B \qquad B <: C}{A <: C} \qquad \textsc{Sub-top}\ \frac{}{A <: \top} \qquad \textsc{Sub-arr}\ \frac{A_2 <: A_1 \qquad B_1 <: B_2}{A_1 \to B_1 <: A_2 \to B_2} \qquad \textsc{Sub-label}\ \frac{A <: B}{A^\alpha <: B^\alpha}$$

$$\textsc{Sub-mu}\ \frac{A[\alpha \mapsto A^\alpha] <: B[\alpha \mapsto B^\alpha]}{\mu\alpha.\, A <: \mu\alpha.\, B} \qquad \textsc{Sub-andl}\ \frac{}{A_1 \,\&\, A_2 <: A_1} \qquad \textsc{Sub-andr}\ \frac{}{A_1 \,\&\, A_2 <: A_2} \qquad \textsc{Sub-and}\ \frac{A <: B_1 \qquad A <: B_2}{A <: B_1 \,\&\, B_2}$$

$$\textsc{Sub-dist-arr}\ \frac{}{(A \to B_1) \,\&\, (A \to B_2) <: A \to (B_1 \,\&\, B_2)} \qquad \textsc{Sub-dist-mu}\ \frac{}{(\mu\alpha.\, A) \,\&\, (\mu\alpha.\, B) <: \mu\alpha.\, (A \,\&\, B)} \qquad \textsc{Sub-top-mu}\ \frac{}{\top <: \mu\alpha.\, \top}$$

$$\textsc{Sub-top-arr}\ \frac{}{\top <: \top \to \top}$$

Fig. 3. Declarative subtyping rules.

Theorem 1 (Completeness of translation subtyping)**.** If $A' \rightsquigarrow A$, $B' \rightsquigarrow B$ and $A' <: B'$, then $A \leq B$

The completeness theorem is relatively easy to prove. Since the translated subtyping system is well-studied, $\leq$ is transitive, so we solve the Sub-trans case.

For case Sub-dist-mu, thanks to polarized subtyping, we can show that

$$\begin{array}{ll} & \mu\alpha.A' \,\&\, \mu\alpha.B' \\ \rightsquigarrow & \zeta(\alpha, l).(A[\alpha^- \mapsto \{\alpha.A\}^l]) \,\&\, \zeta(\alpha, l).(B[\alpha^- \mapsto \{\alpha.B\}^l]) \\ \leq & \zeta(\alpha, l).(A[\alpha^- \mapsto \{\alpha.A\&B\}^l]) \,\&\, \zeta(\alpha, l).(B[\alpha^- \mapsto \{\alpha.A\&B\}^l]) \quad (A\&B \leq A \text{ in polarized subst.}) \\ = & \zeta(\alpha, l).((A\&B)[\alpha^- \mapsto \{\alpha.A\&B\}^l]) \leftsquigarrow \mu\alpha.(A' \,\&\, B') \end{array}$$

For case Sub-mu, we should be able to show that polarized subtyping is sufficient w.r.t. nominal unfolding, with the help of Lemma 2.

Lemma 2 (Polarized substitution to full substitution)**.** If $A[\alpha^- \mapsto \{\alpha.C\}^l] \leq B[\alpha^- \mapsto \{\alpha.D\}^l]$, and $C \leq D$, then $A[\alpha \mapsto \{\alpha.C\}^l] \leq B[\alpha \mapsto \{\alpha.D\}^l]$.

Theorem 3 (Soundness of translation subtyping)**.** If $A' \rightsquigarrow A$, $B' \rightsquigarrow B$ and $A \leq B$, then $A' <: B'$

Proof attempt of Soundness. We prove the soundness of the translation subtyping by induction on the derivation of $A \leq B$ and then inversion on the derivation of $A' \rightsquigarrow A$ and $B' \rightsquigarrow B$.

Most of the cases are straightforward by applying the induction hypothesis. For example, in the case sub-arr, by inversion we know there exists A_1' and A_2' such that $A_1' \rightsquigarrow A_1$ and $A_2' \rightsquigarrow A_2$. By induction hypothesis, we have $A_2' <: A_1'$. Similarly, we can prove $B_2' <: B_1'$. Therefore, by applying the rule Sub-arr, we have $A' \to B'$.

The challenging case is where there are no inversed types, which is the case for sub-and. In this case, when $B' \rightsquigarrow B$ and $B_1 \lhd B \rhd B_2$ are given, B_1 and B_2 are not necessarily guaranteed to be translated from some type B_1' and B_2', so we cannot apply IH.

It is also hard to recover some source type from the splitted target type B_1', B_2', in particular in the case of nested recursive types.

□

3 Intermediate System

We adapt the Siek [2020]'s BCD subtyping system, which keeps the right type invertible throughout the subtyping derivation.

We hope to prove soundness with the help of this intermediate system.

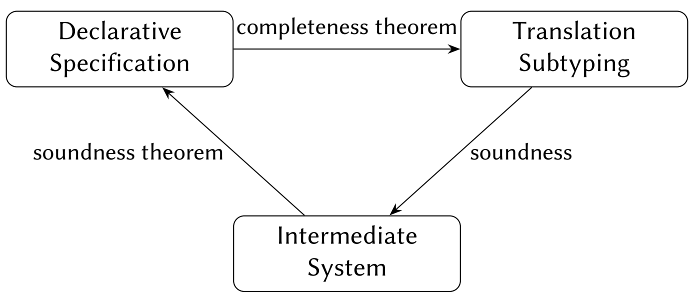

Fig. 4. Structure of the proof

3.1 Containment Relation

$\boxed{A \Subset B}$ *(B contains A)*

$$\frac{}{\mathsf{nat} \Subset \mathsf{nat}}\ \textsc{cont-nat} \qquad \frac{A \Subset B}{A \Subset B \,\&\, C}\ \textsc{cont-andl} \qquad \frac{A \Subset C}{A \Subset B \,\&\, C}\ \textsc{cont-andr} \qquad \frac{}{\alpha \Subset \alpha}\ \textsc{cont-var} \qquad \frac{B \Subset C}{A \rightarrow B \Subset A \rightarrow C}\ \textsc{cont-arr}$$

$$\frac{}{A^\alpha \Subset A^\alpha}\ \textsc{cont-label} \qquad \frac{\alpha \in \mathsf{FV}^-(A)}{\mu\alpha.\,A \Subset \mu\alpha.\,A}\ \textsc{cont-mu-neg} \qquad \frac{\alpha \notin \mathsf{FV}^-(A) \qquad A \Subset B}{\mu\alpha.\,A \Subset \mu\alpha.\,B}\ \textsc{cont-mu-pos}$$

The containment relation treat binary intersections as a sequence. The subsequence relation can be described as $A \underline{\Subset} B$, defined as follows:

$$A \underline{\Subset} B \triangleq \forall C.\, C \Subset A \text{ implies } C \Subset B$$

There are a few properties of the containment relation (which have passed the property based testing). Note: in the testing, we used the translation algorithm $<:_a$ instead of $<:_j$, to avoid exponential blowup of iterating all combinations.

Theorem 4 (Each containment is a supertype)**.** For any two types A and B, if $A \Subset B$, then $B <:_a A$.

Corollary 5. If $A \underline{\Subset} B$, then $B <:_a A$.

Theorem 6 (All containments recover the original type)**.** $(\&_{A_i \Subset B} A_i) <:_a B$

Theorem 7 (Containments can always form an intermediate type)**.** For any two types A and B, if $A <:_a B$, then there exists a type $C \underline{\Subset} A$ such that $C <:_a B$.

A side note on the negative variable testing. Note, we rely on a $\alpha \in \mathsf{FV}^-(A)$ relation to determine whether a variable appears negatively in a type. In this checking we need to take nested recursive types into account. For example, $\alpha \in \mathsf{FV}^-(\mu\beta.\beta \to \alpha)$ is true. In the current implementation, this is achieved by checking the negative occurrences of α in the *translation* of A:

$$\alpha \in \mathsf{FV}^-(A) \triangleq \text{ If } A \rightsquigarrow B, \text{ then } \alpha \in \mathsf{FV}^-(B)$$

but I believe we can define an alternative inductive relation to define $\alpha \in \mathsf{FV}^-(A)$ without relying on the translation.

Another point to note is that we might consider this alternative treatment of $\mathsf{FV}^-(A)$:

$$\text{If} \rceil A \lceil, \text{ then } \alpha \in \mathsf{FV}^-(A) \text{ always holds regardless of whether } \alpha \text{ appears in } A.$$

Since we can always rewrite this toplike type A to $\top$ which contains no α. However, this optimization is not implemented in the current version, and (I presume) might be unnecessary to include in the proof.

3.2 Auxiliary Functions

$\mathsf{dom}(A)$ and $\mathsf{cod}(A)$ are the intersections of all domains and codomains in a function-like type.

$$\begin{array}{lcl} \mathsf{dom}(A \to B) & = & A \\ \mathsf{dom}(A \& B) & = & \mathsf{dom}(A) \& \mathsf{dom}(B) \\ \\ \mathsf{cod}(A \to B) & = & B \\ \mathsf{cod}(A \& B) & = & \mathsf{cod}(A) \& \mathsf{cod}(B) \end{array}$$

They are the same as Siek [2020]'s definitions. In simple BCD settings, A is equivalent to $\mathsf{dom}(A) \to \mathsf{cod}(A)$.

Similarly, we define $\mathsf{mcod}(A)$, which extracts all the codomains of recursive types in A:

$$\begin{array}{lcl} \mathsf{mcod}(\mu\alpha.A) & = & (A) \\ \mathsf{mcod}(A \& B) & = & \mathsf{mcod}(A) \& \mathsf{mcod}(B) \end{array}$$

However, $\mu\alpha.\mathsf{mcod}(A)$ is not equivalent to A, due to negative recursive subtyping. This is also the reason why in the containment relation, we had the overlapping rules CONT-MU-NEG and CONT-MU-POS. For example:

```
mu a. Int -> a      is a containment of     mu a. (Int -> a) & (a -> Int)
mu a. (Int -> a) & (a -> Int)  is also a containment of    mu a. (Int -> a) & (a -> Int)
```

This is to ensure that all the possible minimal components in a recursive type are captured (Theorem 7).

3.3 Subtyping Relation

We first present Siek [2020]'s BCD subtyping rules (without recursive types).

$\boxed{A <:_s B}$ *(Intermediate subtyping system)*

$$\frac{}{\mathsf{nat} <:_s \mathsf{nat}}\ \textsc{jsub-nat} \qquad \frac{\rceil B \lceil}{A <:_s B}\ \textsc{jsub-top} \qquad \frac{A_1 <:_s B}{A_1 \,\&\, A_2 <:_s B}\ \textsc{jsub-andl} \qquad \frac{A_2 <:_s B}{A_1 \,\&\, A_2 <:_s B}\ \textsc{jsub-andr} \qquad \frac{A <:_s B_1 \qquad A <:_s B_2}{A <:_s B_1 \,\&\, B_2}\ \textsc{jsub-and}$$

$$\frac{C <:_s \mathsf{dom}(B) \qquad \mathsf{cod}(B) <:_s D \qquad B \Subset A \qquad \neg\rceil D \lceil}{A <:_s C \to D}\ \textsc{jsub-arr} \qquad \frac{}{\alpha <:_s \alpha}\ \textsc{jsub-var}$$

Fig. 5. Siek [2020]'s BCD subtyping rules without recursive types.

Rule JSUB-ARR is the only non-trivial rule in this system. It deals with function distributive subtyping not by splitting $C \to D$, but by finding a part of A, namely B (which can be regarded as the intersection of several A_i's such that $A_i \Subset A$). And then extract the domain and codomain of B to compare with C and D.

The key characteristic of this rule is that it keeps all the types in the premises invertible (given that the type in the conclusion is invertible), while it still provides a way to destruct the subtyping of function types (by iterating over all components of A).

3.4 Adding Recursive Types to Subtyping Relation - first attempt

We wish to develop a similar subtyping rule for recursive types. The idea is similar – for $A <: \mu\alpha.B$, we find components of A, whose recursive bodies can be merged to form a subtype of $\mu\alpha.B$, so that we can apply the nominal unfolding rule to destruct the recursive types:

$$\frac{C \Subset A \qquad \mathsf{mcod}(C)[\alpha \mapsto \mathsf{mod}(C)^{\alpha}] <:_s B[\alpha \mapsto B^{\alpha}]}{A <:_s \mu\alpha.B}\ \textsc{jsub-mu-attempt}$$

With the rule above we should be able to deal with subtyping relations like:

```
 mu a. (Int -> a) & mu a. (Bool -> a) & mu a. ((a -> Int) & (a -> Bool) )
<: mu a. ((Bool -> a) & (a -> Int) & (a -> Bool))
```

by setting `C = mu a. (Bool -> a) & mu a. ((a -> Int) & (a -> Bool))`. Note that without the $C \Subset A$ condition, we would have to merge all the components of the recursive types in A, which leads to failure in comparing the nominal unfolding due to negative occurrences of α.

It is also helpful at this point to see some examples that our new defined rule CONT-MU-POS and rule CONT-MU-NEG are able to handle. Consider:

```
   mu a. ((Int -> a) & (Bool -> a)) & mu a. ((a -> Int) & (a -> Bool) )
<:  mu a. ((Bool -> a) & (a -> Int) & (a -> Bool))
```

The rule CONT-MU-POS allows us to get two containments `mu a. Int -> a` and `mu a. Bool -> a` from `mu a. ((Int -> a) & (Bool -> a))`, so that the intended C can be formed. By contrast, due to the negative occurrences of α in the second recursive type, the only type it contains is itself (by rule CONT-MU-NEG). Otherwise we get non-equivalent types.

3.5 Adding Recursive Types to Subtyping Relation - refined

However, due to the non-invariant nature of distributing recursive types, rule JSUB-MU-ATTEMPT is not sufficient to handle all the cases. For example, in

```
(mu a. Top -> a) & (mu a. a -> Int) <: mu a. ((Int -> a) & (a -> Int))
```

The subtyping holds in the declarative specification with the help of a middle type:

```
(mu a. Int -> a) & (mu a. a -> Int)
```

However, with the proposed rule JSUB-MU-ATTEMPT, we cannot find a type C for the original type that satisfies the nominal unfolding. The derivation for this example has to be first subtyping on the first left component, and then merge.

To address this issue, we propose to also split on the right, but in a different way than the containment relation does on the left type. The idea is to find splits of $\mu\alpha.B$ such that checking whether A is a subtype of all the types in the split is sufficient to prove $A <: \mu\alpha.B$.

The procedure of finding such splits is described as follows:

(1) Find all the *precise containments* $D_i \Subset' B$.
Note that here we use a different notion of containment than the one used on the left type. Specifically,

$$\Subset' \triangleq \Subset / \textsc{cont-mu-pos} \cup \textsc{cont-mu-pos-strict}$$

. The replaced rule CONT-MU-POS-STRICT is as follows:

$$\textsc{cont-mu-pos-strict}\quad \frac{\alpha \notin \mathsf{FV}^-(B) \qquad A \Subset B}{\mu\alpha.\,A \Subset \mu\alpha.\,B}$$

Unlike the original rule CONT-MU-POS, which allows one to consider a positive part of A to be the containment of B even if B is negative, e.g.:

```
mu a. Int -> a      is a containment of      mu a. (Int -> a) & (a -> Int)
```

Here since we are splitting on the right, we want the containment to be precise, we simply want `mu a. (Int -> a) & (a -> Int)` to be a *precise* containment of itself.

(2) We sort out all the positive containments D_i^{pos} and negative containments D_j^{neg} from the list of precise containments, based on whether $\alpha \in \mathsf{FV}^-(D_i)$ or not. Then the positive containments D_i^{pos} are collected as a list $\mu\alpha.D_i^{pos}$, while the negative containments are merged into a single type $\mu\alpha.(\&_{\forall j}\, D_j^{neg})$.
We write $\alpha \vdash C_1, \ldots, C_n \trianglerighteq A \trianglelefteq B$ to denote that recursive type $\mu\alpha.A$ is merged from a negative $\mu\alpha.B$ and several positive $\mu\alpha.C_i$'s.
Figure 6 shows an example of this splitting.

Original type:

```
mu a.
    (Int -> a)
    & (a -> Int)
    & (mu b.
            (Int -> a)
        & (Int -> b))
    & (mu b.
            (Bool -> b)
        & (b -> Int))
```

Splits into:

```
Positive parts:
mu a. Int -> a
mu a. mu b. Int -> a
mu a. mu b. Int -> b
mu a. mu b. ((Bool -> b) & (b -> Int))

Negative part:
mu a.(
    (a -> Int)
    & (mu b.
        (Bool -> b)
        & (b -> Int))
)
```

Fig. 6. Example of right splitting

(3) Rule JSUB-MU is the final rule for subtyping on recursive types. It applys the original containment relation on the left type, and the new precise containment finding procedure we proposed above on the right type.
Then, it divides the positive parts into two groups, one $D_1, \ldots, D_n$ that are compared directly with the nominal unfolding of $\mathrm{mod}(C)$, and the other $D'_1, \ldots, D'_k$ that are merged into the negative part and compared as a whole with the nominal unfolding of $\mathrm{mod}(C)$.

$\boxed{A <:_s B}$ *(Intermediate subtyping system)*

JSUB-MU

$$\frac{\begin{array}{c} C \Subset A \qquad \alpha \vdash D_1,\, ..,D_n, D'_1,\, ..,D'_k \trianglerighteq B \trianglelefteq S \qquad \neg\rceil \mu\alpha.\, B \lceil \\ \mathsf{mcod}(C)[\alpha \mapsto \mathsf{mcod}(C)^{\alpha}] <:_s D_1[\alpha \mapsto {D_1}^{\alpha}] \quad .. \quad \mathsf{mcod}(C)[\alpha \mapsto \mathsf{mcod}(C)^{\alpha}] <:_s D_n[\alpha \mapsto {D_n}^{\alpha}] \\ \mathsf{mcod}(C)[\alpha \mapsto \mathsf{mcod}(C)^{\alpha}] <:_s ((D'_1 \& .. \& D'_k) \,\&\, S)[\alpha \mapsto ((D'_1 \& .. \& D'_k) \,\&\, S)^{\alpha}] \end{array}}{A <:_s \mu\alpha.\, B}$$

Fig. 7. Siek [2020]'s BCD subtyping rules extended with recursive types.

Here are some remarks on the rule JSUB-MU rule:

- Despite backtracking on a lot of combinations of the containments on both sides, the rule still ensures that the depth of recursive types is decreasing (maybe no, since we use nominal unfolding, but we should have ways to adopt some techniques before to deal with that) in the derivation. So we should have a well-founded induction principle / size function to reason about this rule.
- It should be straightforward to see the soundness of this intermediate system to the declarative specification. We can turn all the containments into subtyping relations (both in rules JSUB-MU and JSUB-ARR), and then apply the transitivity rule.
- The soundness of the translation algorithm to this intermediate system might be more non-trivial. But the advantage of this intermediate system is that rule JSUB-MU only deals with one level of recursive types, and it delegates the subtyping of inner recursive binders to the nominal unfolding rule. So compared to the translation algorithm, whose split operation can be arbitrarily deep, across several recursive binders, the reasoning on nested recursion for this rule is expected to be simpler.

References

Xuejing Huang, Jinxu Zhao, and Bruno C. D. S. Oliveira. 2021. Taming the Merge Operator. *Journal of Functional Programming* 31 (Jan. 2021), e28. doi:10.1017/S0956796821000186

Jeremy G. Siek. 2020. Transitivity of Subtyping for Intersection Types. doi:10.48550/arXiv.1906.09709 arXiv:1906.09709 [cs]