Types	T	$::=$	α	Type variable
		$\mid$	$\bot$	Bottom type
		$\mid$	$A \to B$	Function type
		$\mid$	$\forall \alpha * B.\, A$	Universal quantification
		$\mid$	$A \cap B$	Intersection type
Expressions	e	$::=$	x	Variable
		$\mid$	$\lambda(x{:}A).\, e$	Lambda
		$\mid$	$e_1\ e_2$	Application
		$\mid$	$\Lambda \alpha * A.\, e$	Big lambda
		$\mid$	$e\ A$	Type application
		$\mid$	$e_1 ,, e_2$	Merge
Contexts	Γ	$::=$	ϵ	
		$\mid$	$\Gamma, \alpha * A$	
		$\mid$	$\Gamma, x{:}A$	
Sugar	$\Lambda \alpha.\, e$	$::=$	$\Lambda \alpha * \bot.\, e$	

Figure 1. Syntax.

$$\boxed{A <: B \hookrightarrow F}$$

$$\frac{}{\alpha <: \alpha \hookrightarrow \lambda(x{:}|\alpha|).\, x}\ \textsc{SubVar}$$

$$\frac{\tau_3 <: \tau_1 \hookrightarrow C_1 \qquad \tau_2 <: \tau_4 \hookrightarrow C_2}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4 \hookrightarrow \lambda(f{:}|\tau_1 \to \tau_2|).\, \lambda(x{:}|\tau_3|).\, C_2\ (f\ (C_1\ x))}\ \textsc{SubFun}$$

$$\frac{\tau_1 <: [\alpha_1/\alpha_2]\tau_2 \hookrightarrow C}{\forall \alpha_1 * \tau_3.\, \tau_1 <: \forall \alpha_2 * \tau_3.\, \tau_2 \hookrightarrow \lambda(f{:}|\forall \alpha_1 * \tau_3.\, \tau_1|).\, \Lambda \alpha.\, C\ (f\ \alpha)}\ \textsc{SubForall}$$

$$\frac{\tau_1 <: \tau_2 \hookrightarrow C_1 \qquad \tau_1 <: \tau_3 \hookrightarrow C_2}{\tau_1 <: \tau_2 \cap \tau_3 \hookrightarrow \lambda(x{:}|\tau_1|).\, (C_1\ x, C_2\ x)}\ \textsc{SubAnd}$$

$$\frac{\tau_1 <: \tau_3 \hookrightarrow C}{\tau_1 \cap \tau_2 <: \tau_3 \hookrightarrow \lambda(x{:}|\tau_1 \cap \tau_2|).\, C\ (\mathsf{proj}_1 x)}\ \textsc{SubAnd}_1$$

$$\frac{\tau_2 <: \tau_3 \hookrightarrow C}{\tau_1 \cap \tau_2 <: \tau_3 \hookrightarrow \lambda(x{:}|\tau_1 \cap \tau_2|).\, C\ (\mathsf{proj}_2 x)}\ \textsc{SubAnd}_2$$

[1]

Figure 2. Subtyping.

0.1 "Testsuite" of examples

1. $\lambda(x : \mathrm{Int} * \mathrm{Int}).(\lambda(z : \mathrm{Int}).z)\ x$: This example should not type-check because it leads to an ambigous choice in the body of the lambda. In the current system the well-formedness checks forbid such example.
2. $\Lambda A.\Lambda B.\lambda(x : A).\lambda(y : B).(\lambda(z : A).z)(x,,y)$: This example should not type-check because it is not guaranteed that the instantiation of A and B produces a well-formed type. The TyMerge rule forbids it with the disjointness check.

$$\boxed{A\ \mathtt{atomic}}$$

$$\bot\ \mathtt{atomic} \qquad A \to B\ \mathtt{atomic} \qquad \forall \alpha * B.\, A\ \mathtt{atomic}$$

Figure 3. Atomic types.

$$\boxed{\Gamma \vdash A \bot B}$$

$$\frac{\alpha * B \in \Gamma}{\Gamma \vdash \alpha \bot B}\ \textsc{DisjointRefl} \qquad \frac{\alpha * A \in \Gamma}{\Gamma \vdash A \bot \alpha}\ \textsc{DisjointSym}$$

$$\frac{\Gamma \vdash A \bot C \qquad \Gamma \vdash B \bot C}{\Gamma \vdash A \& B \bot C}\ \textsc{DisjointSub1}$$

$$\frac{\Gamma \vdash A \bot B \qquad \Gamma \vdash A \bot C}{\Gamma \vdash A \bot B \& C}\ \textsc{DisjointSub2}$$

$$\frac{\begin{array}{ccc} A \not<: B & & \\ B \not<: A & A\ \mathtt{atomic} & B\ \mathtt{atomic} \end{array}}{\Gamma \vdash A \bot B}\ \textsc{DisjointAtomic}$$

Figure 4. Disjointness.

$$\boxed{\Gamma \vdash A\ \mathtt{type}}$$

$$\Gamma \vdash \bot\ \mathtt{type}\ \textsc{WFBot} \qquad \frac{\Gamma \vdash A\ \mathtt{type} \qquad \Gamma \vdash B\ \mathtt{type}}{\Gamma \vdash A \to B\ \mathtt{type}}\ \textsc{WFFun}$$

$$\frac{\Gamma, \alpha * B \vdash A\ \mathtt{type}}{\Gamma \vdash \forall \alpha * B.\, A\ \mathtt{type}}\ \textsc{WFForall} \qquad \frac{\alpha * A \in \Gamma}{\Gamma \vdash \alpha\ \mathtt{type}}\ \textsc{WFVar}$$

$$\frac{\Gamma \vdash A\ \mathtt{type} \qquad \Gamma \vdash B\ \mathtt{type} \qquad \Gamma \vdash A \bot B}{\Gamma \vdash A \cap B \mathtt{type}}\ \textsc{WFInter}$$

Figure 5. Well-formed types.

3. $\Lambda A.\Lambda B * A.\lambda(x : A).\lambda(y : B).(\lambda(z : A).z)(x,,y)$: This example should type-check because B is guaranteed to be disjoint with A. Therefore instantiation should produce a well-formed type.
4. $(\lambda(z : \mathrm{Int}).z)((1,,'c'),,(2,\mathtt{False}))$: This example should not type-check, since it leads to an ambigous lookup of integers (can either be 1 or 2). The definition of disjointness is crutial to prevent this example from type-checking. When type-checking the large merge, the disjointness predicate will detect that more than one integer exists in the merge.

0.2 Achieving Coherence

The crutial challenge lies in the generation of coercions, which can lead to different results due to multiple possible choices in the rules that can be used. In particular the rules SubAnd1 and SubAnd2 overlap and can result in coercions that are not equivalent. A simple example is:

$(\lambda(x : \mathrm{Int}).x)(1,,2)$

The result of this program can be either 1 or 2 depending on whether we chose SubAnd1 or SubAnd2.

$$\boxed{\Gamma \vdash e : A \hookrightarrow E}$$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A \hookrightarrow x}\ \textsc{TyVar}$$

$$\frac{\Gamma \vdash A\ \text{type} \qquad \Gamma, x{:}A \vdash e : B \hookrightarrow E}{\Gamma \vdash \lambda(x{:}A).\, e : A \to B \hookrightarrow \lambda(x{:}|A|).\, E}\ \textsc{TyLam}$$

$$\frac{\Gamma \vdash e_1 : A_1 \to A_2 \hookrightarrow E_1 \qquad \Gamma \vdash e_2 : A_3 \hookrightarrow E_2 \qquad A_3 <: A_1 \hookrightarrow C}{\Gamma \vdash e_1\ e_2 : A_2 \hookrightarrow E_1\ (C\ E_2)}\ \textsc{TyApp}$$

$$\frac{\Gamma, \alpha * B \vdash e : A \hookrightarrow E \qquad \Gamma \vdash B\ \text{type}}{\Gamma \vdash \Lambda\alpha * B.\, e : \forall \alpha * B.\, A \hookrightarrow \Lambda\alpha.\, E}\ \textsc{TyBLam}$$

$$\frac{\Gamma \vdash e : \forall \alpha * C.\, B \hookrightarrow E \qquad \Gamma \vdash A \bot C \qquad \Gamma \vdash A\ \text{type}}{\Gamma \vdash e\ A : [A/\alpha]B \hookrightarrow E\ |A|}\ \textsc{TyTApp}$$

$$\frac{\Gamma \vdash e_1 : A \hookrightarrow E_1 \qquad \Gamma \vdash e_2 : B \hookrightarrow E_2 \qquad \Gamma \vdash A \bot B}{\Gamma \vdash e_1 ,, e_2 : A \cap B \hookrightarrow (E_1, E_2)}\ \textsc{TyMerge}$$

Figure 6. Typing.

Therefore the challenge of coherence lies in ensuring that, for any given types A and B, the result of $A <: B$ always leads to the same (or semantically equivalent) coercions.

It is clear that, in general, the following does not hold:

$\mathit{if}\ A <: B \rightsquigarrow C1\ \mathit{and}\ A <: B \rightsquigarrow C2\ \mathit{then}\ C1 = C2$

We can see this with the example above. There are two possible coercions:

$(\mathtt{Int\&Int}) <: \mathtt{Int} \rightsquigarrow \lambda(x, y).x$
$(\mathtt{Int\&Int}) <: \mathtt{Int} \rightsquigarrow \lambda(x, y).y$

However $\lambda(x, y).x$ and $\lambda(x, y).y$ are not semantically equivalent.

One simple observation is that the use of the subtyping relation on the example uses an ill-formed type ($\mathtt{Int\&Int}$). Since the type system can prevent such bad uses of ill-formed types, it could be that if we only allow well-formed types then the uses of the subtyping relation do produce equivalent coercions. Therefore the we postulate the following conjecture:

$\mathit{if}\ A <: B \rightsquigarrow C1\ \mathit{and}\ A <: B \rightsquigarrow C2\ \mathit{and}\ A, B\ \mathit{well\ formed\ then}\ C1 = C2$

If the following conjecture does hold then it should be easy to prove that the translation is coherent.

$$\epsilon \vdash 1 ,, 2 : (\mathsf{Int} * \mathsf{Int}) \Rightarrow \mathsf{Int} \cap \mathsf{Int}$$

Definition 1. (Disjointness) Two sets S and T are *disjoint* if there does not exist an element x, such that $x \in S$ and $x \in T$.

Definition 2. (Disjointness) Two types A and B are *disjoint* if there does not exist an expression e, which is not a merge, such that $\epsilon \vdash e : A'$, $\epsilon \vdash e : B'$, $A' <: A$, and $B' <: B$.

Definition 3. (Disjointness) $A \bot B = \nexists C. A <: C \wedge B <: C$

Two types A and B are *disjoint* if their least common supertype is $\top$.