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	$\Lambda \alpha.\, e$	$\equiv$	$\Lambda \alpha * \bot.\, e$	
	$e : A$	$\equiv$	$(\lambda z : A.z) e$	

Figure 1. Syntax.

1. Introduction

Dundfield's work showed how many language features can be encoded in terms of intersection types with a merge operator. However two important questions were left open by Dundfield:

1. How to allow coherent programs only?
2. If a restriction that allows coherent programs is in place, can all coherent programs conform to the restriction?

In other words question 1) asks whether we can find sufficient conditions to guarantee coherency; whereas question 2) asks whether those conditions are also necessary. In terms of technical lemmas that would correspond to:

1. Coherency theorem: $\Gamma \vdash e : A \rightsquigarrow E_1 \wedge \Gamma \vdash e : A \rightsquigarrow E_2 \to E_1 = E_2$.
2. Completness of Coherency: $(\Gamma \vdash_{old} e : A \rightsquigarrow E_1 \wedge \Gamma \vdash_{old} e : A \rightsquigarrow E_2 \to E_1 = E_2) \to \Gamma \vdash e : A$.

For these theorems we assume two type systems. On liberal type system that ensures type-safety, but not coherence ($\Gamma \vdash_{old} e : A$); and another one that is both type-safe and coherent ($\Gamma \vdash e : A$). What needs to be shown for completness is that if a coherent program type-checks in the liberal type system, then it also type-checks in the restricted system.

1.1 "Testsuite" of examples

1. $\lambda(x : \mathtt{Int} * \mathtt{Int}).(\lambda(z : \mathtt{Int}).z)\ x$: 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.
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. $(\lambda(z : \mathtt{Int}).z)((1,,'c'),,(2,\mathtt{False}))$: 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.
5. $(\lambda(f : \mathtt{Int} \to \mathtt{Int\&Bool}).\lambda(g : \mathtt{Int} \to \mathtt{Char\&Bool}).((f,,g) : \mathtt{Int} \to \mathtt{Bool})$: This example should not type-check, since it leads to an ambigous lookup of functions. It shows that in order to check disjointness of functions we must also check disjointness of the subcomponents.
6. $(\lambda(f : \mathtt{Int} \to \mathtt{Int}).\lambda(g : \mathtt{Bool} \to \mathtt{Int}).((f,,g) : \mathtt{Bool\&Int} \to \mathtt{Int})$: This example shows that whenever the return types overlap, so does the function type: we can always find a common subtype for the argument types.

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

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

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

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

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

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

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

[1]

Figure 2. Subtyping.

1.2 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:

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

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

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:

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

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

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

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

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

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

$$\frac{\Gamma \vdash B \perp D}{\Gamma \vdash A \to B \perp C \to D}\ \textsc{DisjointFun}$$

$$\frac{\Gamma \vdash A \perp C}{\Gamma \vdash \forall \alpha * B.\, A \perp \forall \alpha * B.\, C}\ \textsc{DisjointForall}$$

$$\frac{A \not\sim B}{\Gamma \vdash A \perp B}\ \textsc{DisjointAtomic}$$

$$\boxed{A \not\sim B}$$

$$\perp \not\sim A \to B\ \textsc{NotSimBot1} \qquad \perp \not\sim \forall \alpha * B.\, A\ \textsc{NotSimBot2}$$

$$A \to B \not\sim \forall \alpha * B.\, A\ \textsc{NotSimFunForall}$$

$$\frac{B \not\sim A}{A \not\sim B}\ \textsc{NotSimFunForall}$$

Figure 3. Disjointness.

$$\boxed{\Gamma \vdash \tau}$$

$$\frac{\alpha * A \in \Gamma}{\Gamma \vdash \alpha}\ \textsc{WFVar} \qquad \frac{}{\Gamma \vdash \bot}\ \textsc{WFBot}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \textsc{WFFun}$$

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

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad \Gamma \vdash A \perp B}{\Gamma \vdash A \cap B}\ \textsc{WFInter}$$

Figure 4. Well-formed types.

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

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

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

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

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

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

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

Figure 5. Typing.

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

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 ($\mathsf{Int}\&\mathsf{Int}$). 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:

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

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

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

We say two types are *disjoint* if they do not share a common supertype.

Definition 1 (Disjointness). $A \perp B = \nexists C. A <: C \wedge B <: C$

We require the types of two terms in a merge $e_1 ,, e_2$ to be disjoint. Why do we require this? That is because if both terms can be assigned some type C, both of them can be chosen as the meaning of the merge, which leads to multiple meaning of a term, known as incoherence.

2. Polymorphism with disjoint constraint

With a subtyping relation in a type system, bounded polymorphism extends the universal quantifier by confining the polymorphic type to be a subtype of a given type. In our type system, the forall binder

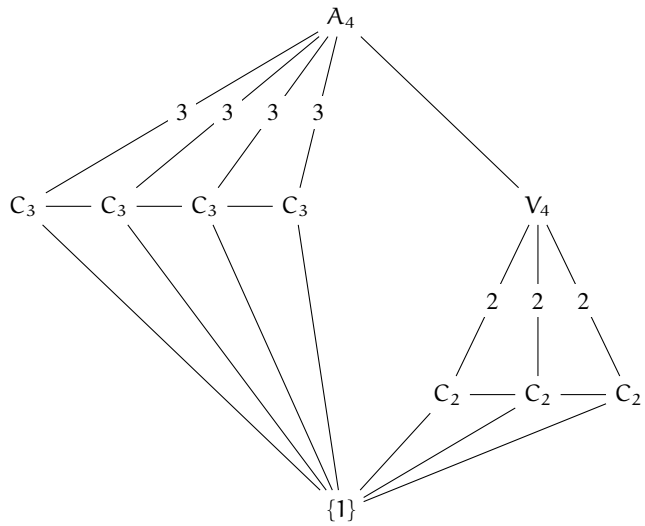

Figure 6. Untergruppenverband

also extends the parametric polymorphism, but in a different vein: the polymorphic type can only be disjoint with a given type.

- **Bounded polymorphism**—the instantiation can only be the descendant of a given type
- **Polymorphism with disjoint constraint**—the instantiation cannot share a common ancestor with a given type

The intuition can be found in figure ....

3. Intuition for the disjoint rules

The problem with the definition of disjointness is that it is a search problem. In this section, we are going to convert it that into an algorithm.

Let $\mathbb{U}_0$ be the universe of τ types. Let $\mathbb{U}$ be the quotient set of $\mathbb{U}_0$ by $\approx$, where $\approx$ is defined by ....

Let $\Uparrow$ be the "common supertype" function, and $\Downarrow$ be the "common subtype" function. For example, assume Int and Char share no common supertype. Then the fact can be expressed by $\Uparrow(\mathsf{Int}, \mathsf{Char}) = \emptyset$. Formally,

$$\Uparrow : \mathbb{U} \times \mathbb{U} \to \mathcal{P}(\mathbb{U})$$
$$\Downarrow : \mathbb{U} \times \mathbb{U} \to \mathcal{P}(\mathbb{U})$$

which, given two types, computes the set of their common supertypes. ($\mathcal{P}(S)$ denotes the power set of S, that is, the set of all subsets of S.)

$$\Uparrow(\alpha, \alpha) = \{\alpha\}$$
$$\Uparrow(\bot, \bot) = \{\bot\}$$
$$\Uparrow(\tau_1 \to \tau_2, \tau_3 \to \tau_4) = \Downarrow(\tau_1, \tau_3) \to \Uparrow(\tau_2, \tau_4)$$

Notation. We use $\Downarrow(\tau_1, \tau_3) \to \Uparrow(\tau_2, \tau_4)$ as a shorthand for $\{s \to t \mid s \in \Downarrow(\tau_1 \to \tau_2), t \in \Uparrow(\tau_2, \tau_4)\}$. Therefore, the problem of determining if $\Downarrow(\tau_1, \tau_3) \to \Uparrow(\tau_2, \tau_4)$ is empty reduces to the problem of determining if $\Uparrow(\tau_2, \tau_4)$ is empty.

Note that there always exists a common subtype of any two given types (case disjoint / case nondisjoint).

4. Proof

4.1 Sketch of the proof

Lemma 1. *If* $A <: B$ *where both* A *and* B *are well-formed, then* A *and* B *cannot be disjoint.*

Proof. $A <: B$ implies B is a common supertype of A and B. As a result, A and B are not disjoint by definition. □

Lemma 2 (Unique subtype contributor)**.** *If* $A \cap B <: C$*, where* $A \cap B$ *and* C *are well-formed types, then it is not possible that the following hold at the same time:*

1. $A <: C$
2. $B <: C$

If $A \cap B <: C$, then either A or B contributes to that subtyping relation, but not both. The implication of this lemma is that during the derivation, it is not possible that two rules are applicable.

Proof. Since $A \cap B$ is well-formed, $A * B$ by the formation rule of intersection types WFINTER. Then by the definition of disjointness, there does not exist a type C such that $A <: C$ and $B <: C$. It follows that $A <: C$ and $B <: C$ cannot hold simultaneously. □

The coercion of a subtyping relation $A <: B$ is uniquely determined.

Lemma 3 (Unique coercion)**.** *If* $A <: B \hookrightarrow C_1$ *and* $A <: B \hookrightarrow C_2$*, where* A *and* B *are well-formed types, then* $C_1 \equiv C_2$

Proof. The set of rules for generating coercions is syntax-directed except for the three rules that involve intersection types in the conclusion. Therefore it suffices to show that if well-formed types A and B satisfy $A <: B$, where A or B is an intersection type, then at most one of the three rules applies. In the following, we do a case analysis on the shape of A and B:

- **Case** $A \neq A_1 \cap A_2$ and $B = B_1 \cap B_2$: Clearly only SUBAND can apply.
- **Case** $A = A_1 \cap A_2$ and $B \neq B_1 \cap B_2$: Only two rules can apply, SUBAND1 and SUBAND2. Further, by the unique subtype contributor lemma, it is not possible that $A_1 <: B$ and that $A_2 <: B$. Thus we are certain that at most one rule of SUBAND1 and SUBAND2 will apply.
- **Case** $A = A_1 \cap A_2$ and $B = B_1 \cap B_2$[2]: Since B is not atomic, only SubAnd apply.

□

A naive substitution can violate the disjoint constraint in the context. For example (assuming the existence of some base type Int),

$$\frac{\alpha * \mathsf{Int} \in \alpha * \mathsf{Int}}{\alpha * \mathsf{Int} \vdash \alpha}\ \textsc{Foo}$$

is provable by .... But after the substitution of Int for α, ...

$$\alpha * \mathsf{Int} \vdash [\mathsf{Int}/\alpha]\ \alpha$$

Lemma 4. *If* $\Gamma \vdash A \bot B$ *and the* R, γ *pair does not violate any of the disjointness constraints in* Γ*, then* $\Gamma \vdash [R/\gamma]\ A \bot [R/\gamma]\ B$.

Lemma 5. *Substitution lemma If* $\Gamma \vdash R$, $\Gamma \vdash S$, *and the* R, γ *pair does not violate any of the disjointness constraints in* Γ*, then* Elaborate

[2] An example of this case is:

$$(\mathsf{Int} \cap \mathsf{Bool}) \cap \mathsf{Char} <: \mathsf{Bool} \cap \mathsf{Char}$$

$\Gamma \vdash [R/\beta]\ S$.

Proof. By induction on the derivation of $\Gamma \vdash [R/\beta]\ S$.

- Case
$$\frac{\alpha * A \in \Gamma}{\Gamma \vdash \alpha}\ \textsc{WFVar}$$
If α happens to be the same as β, then by the def. of substitution $[R/\beta]\ \alpha = R$. Since $\Gamma \vdash R$, we have $\Gamma \vdash [R/\beta]\ \alpha$; On the other hand, if not, then by the def. of substitution $[R/\beta]\ S = S$. Since $\Gamma \vdash S$, we also have $\Gamma \vdash [R/\beta]\ \alpha$.
- Case
$$\frac{}{\Gamma \vdash \bot}\ \textsc{WFBot}$$
Trivial.
- Case
$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \textsc{WFFun}$$
By i.h., $\Gamma \vdash [R/\gamma]\ A$ and $\Gamma \vdash [R/\gamma]\ B$. By the def. of substitution, $\Gamma \vdash [R/\gamma]\ A \to B$.
- Case
$$\frac{\Gamma \vdash A \qquad \Gamma, \alpha * A \vdash B}{\Gamma \vdash \forall \alpha * A.\ B}\ \textsc{WFForall}$$
By i.h., $\Gamma \vdash [R/\gamma]\ A$ and $\Gamma, \alpha * A \vdash [R/\gamma]\ B$. By the def. of substitution, $\Gamma \vdash [R/\gamma]\ \forall \alpha * A.\ B$.

How about A?

- Case
$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad \Gamma \vdash A \bot B}{\Gamma \vdash A \cap B}\ \textsc{WFInter}$$
By i.h., $\Gamma \vdash [R/\gamma]\ A$ and $\Gamma \vdash [R/\gamma]\ B$. By some magic lemma , we also have $\Gamma \vdash [R/\gamma]\ A \bot [R/\gamma]\ B$. Therefore by (WFInter), $\Gamma \vdash [R/\gamma]\ A \cap B$.

TODO

□

Lemma 6. *Well-formed typing If* $\Gamma \vdash e : \tau \hookrightarrow E$ *, then* τ *is a well-formed type.*

Proof. Proof by induction on the derivation.

The special case to consider is (TyTApp). $[A/\alpha]\ B$ is also well-formed by **??**. □

Given a source expression e, elaboration always produces the same target expression E.

Theorem 1 (Unique elaboration)**.** *If* $\Gamma \vdash e : \tau_1 \hookrightarrow E_1$ *and* $\Gamma \vdash e : \tau_2 \hookrightarrow E_2$ *, then* $E_1 \equiv E_2$

Proof. The typing rules are syntax-directed. It suffices to show that C in TyApp is unique. Note that A_3 and A_1 are well-formed due to Lemma 6. Therefore C is unique. □

5. Application of the theory

5.1 Systems without subtyping

5.2 Systems with a top type

In type systems with a top type (such as `Object` in some OO languages), the definition of disjointness can be modified to:

We say two types are *disjoint* if their only common supertype is the top type.