Disjoint Intersection Types and Disjoint Quantification

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

Over the years there have been various proposals for *design patterns* to improve *extensibility* of programs. Examples include *Object Algebras*, *Modular Visitors* or Torgersen's design patterns using generics. Although those design patterns give practical benefits in terms of extensibility, they also expose limitations in existing mainstream OOP languages. Some pressing limitations are: 1) lack of good mechanisms for *object-level* composition; 2) *conflation of (type) inheritance with subtyping*; 3) *heavy reliance on generics*.

This paper presents System $\mathsf{F}_{\&}$: an extension of System F with *intersection types* and a *merge operator*. The goal of System $\mathsf{F}_{\&}$ is to study the minimal language constructs needed to support various extensible designs, while at the same time addressing the limitations of existing OOP languages. To address the lack of good object-level composition mechanisms, System $\mathsf{F}_{\&}$ uses the merge operator to do dynamic composition of values/objects. Moreover, in System $\mathsf{F}_{\&}$ type inheritance is independent of subtyping, and an extension can be a supertype of a base object type. Finally, System $\mathsf{F}_{\&}$ replaces many uses of generics by intersection types or conventional subtyping. System $\mathsf{F}_{\&}$ is formalized and implemented. Moreover the paper shows how various extensible designs can be encoded in System $\mathsf{F}_{\&}$.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms Design, Languages, Theory

Keywords Intersecion Types, Polymorphism, Type System

1. Introduction

There has been a remarkable number of works aimed at improving support for extensibility in programming languages. The motivation behind this line of work is simple, and it is captured quite elegantly by the infamous *Expression Problem* [39]: there are *two* common and desirable forms of extensibility, but most mainstream languages can only support one form well. Unfortunately the lack of support in the other form has significant consequences in terms of code maintenance and software evolution. As a result researchers proposed various approaches to address the problem, including: visions of new programming models [19, 31, 36]; new programming languages or language extensions [3, 22, 24, 35], and *design patterns* that can be used with existing mainstream languages [10, 25, 37, 40].

Some of the more recent work on extensibility is focused on design patterns. Examples include *Object Algebras* [25], *Modular Visitors* [10, 37] or Torgersen's [37] four design patterns using generics. In those approaches the idea is to use some advanced (but already available) features, such as *generics* [4], in combination with conventional OOP features to model more extensible designs. Those designs work in modern OOP languages such as Java, C#, or Scala.

Although such design patterns give practical benefits in terms of extensibility, they also expose limitations in existing mainstream OOP languages. In particular there are three pressing limitations: 1) lack of good mechanisms for *object-level* composition; 2) *conflation of (type) inheritance with subtyping*; 3) *heavy reliance on generics*.

The first limitation shows up, for example, in encodings of Feature-Oriented Programming [31] or Attribute Grammars [21] using Object Algebras [26, 32]. These programs are best expressed using a form of *type-safe*, *dynamic*, *delegation*-based composition. Although such form of composition can be encoded in languages like Scala, it requires the use of low-level reflection techniques, such as dynamic proxies, reflection or other forms of metaprogramming. It is clear that better language support would be desirable.

The second limitation shows up in designs for modelling modular or extensible visitors [10, 37]. The vast majority of modern OOP languages combines type inheritance and subtyping. That is, a type extension induces a subtype. However as Cook et al. [8] famously argued there are programs where "*subtyping is not inheritance*". Interestingly not many programs have been previously reported in the literature where the distinction between subtyping and inheritance is relevant in practice. However, as shown in this paper, it turns out that this difference does show up in practice when designing modular (extensible) visitors. We believe that modular visitors provide a compelling example where inheritance and subtyping should not be conflated!

Finally, the third limitation is prevalent in many extensible designs [10, 26, 32, 37, 40]. Such designs rely on advanced features of generics, such as *F-bounded polymorphism* [5], *variance annotations* [20], *wildcards* [38] and/or *higher-kinded types* [23] to achieve type-safety. Sadly, the amount of type-annotations, combined with the lack of understanding of these features, usually deters programmers from using such designs.

This paper presents System $\mathsf{F}_{\&}$ (pronounced *f-and*): an extension of System F [33] with intersection types and a merge operator [14]. The goal of System $\mathsf{F}_{\&}$ is to study the *minimal* foundational language constructs that are needed to support various extensible designs, while at the same time addressing the limitations of existing OOP languages. To address the lack of good object-level composition mechanisms, System $\mathsf{F}_{\&}$ uses the merge operator for dynamic composition of values/objects. Moreover, in System $\mathsf{F}_{\&}$ (type-level) extension is independent of subtyping, and it is possible for an extension to be a supertype of a base object type. Furthermore, intersection types and conventional subtyping can be used in many cases instead of advanced features of generics. Indeed this paper shows many previous designs in the literature can be encoded without such advanced features of generics.

Technically speaking System $\mathsf{F}_{\&}$ is mainly inspired by the work of Dundfield [14]. Dundfield showed how to model a simply typed calculus with intersection types and a merge operator. The presence of a merge operator adds significant expressiveness to the lan-

guage, allowing encodings for many other language constructs as syntactic sugar. System $F_{\&}$ differs from Dundfield's work in a few ways. Firstly, it adds parametric polymorphism and formalizes an extension for records to support a basic form of objects. Secondly, the elaboration semantics into System F is done directly from the source calculus with subtyping. Finally, a non-technical difference is that System $F_{\&}$ is aimed at studying issues of OOP languages and extensibility, whereas Dunfield's work was aimed at Functional Programming and he did not consider application to extensibility. Like many other foundational formal models for OOP (for example $F_{<:}$ [6]), System $F_{\&}$ is purely functional and it uses structural typing.

In summary, the contributions of this paper are:

- **A Minimal Core Language for Extensibility:** This paper identifies a minimal core language, System $F_{\&}$, capable of expressing various extensibility designs in the literature. System $F_{\&}$ also addresses limitations of existing OOP languages that complicate extensible designs.
- **Formalization of System $F_{\&}$:** An elaboration semantics of System $F_{\&}$ into System F is given, and type-soundness is proved.
- **Encodings of Extensible Designs:** Various encodings of extensible designs into System $F_{\&}$, including *Object Algebras* and *Modular Visitors*.
- **A Practical Example where "Inheritance is not Subtyping" Matters:** This paper shows that modular/extensible visitors suffer from the "inheritance is not subtyping problem".
- **Implementation:** An implementation of an extension of System $F_{\&}$, as well as the examples presented in the paper, are publicly available[1].

2. Overview

This section introduces $F_{\&}$ and its support for intersection types and polymorphism. It then shows that, without care, the system lacks *coherence*. Finally it is shown that by allowing only disjoint intersection types and extending universal quatification to disjoint quantification, coherence is possible.

2.1 Intersection Types

BRUNO: First show what intersection types are and why they are useful

2.2 Parametric Polymorphism

BRUNO: Then talk about parametric polymorphism

2.3 Intersection Types and Noncoherence

What is an intersection type? The intersection of types A and B contains exactly those values which can be used as either of type A or of type B. Just as not all intersection of sets are nonempty, not all intersections of types are inhabited. For example, the intersection of a base type Int and a function type $\mathsf{Int} \to \mathsf{Int}$ is not inhabited. The merge operator combines two terms, of type A and B respectively, to form a term of type $A \cap B$. For example, $1,,\text{'c'}$ is of type $\mathsf{Int} \cap \mathsf{Char}$. In this case, no matter $1,,\text{'c'}$ is used as Int or Char, the result of evaluation is always clear. However, with overlapping types, it is not straightforward anymore to see the result. For example, what should be the result of this program, which asks for an integer out of a merge of two integers:

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

Should the result be 1 or 2?

[1] **Note to reviewers:** Due to the anonymous submission process, the code (and some machine checked proofs) is submitted as supplementary material.

The following shows the naive subtyping rules for intersection types:

$$\frac{A_1 <: A_2 \hookrightarrow C_1 \qquad A_1 <: A_3 \hookrightarrow C_2}{A_1 <: A_2 \cap A_3 \hookrightarrow \lambda x{:}|A_1|.\ (C_1\ x, C_2\ x)}\ \textsc{SubAnd}$$

$$\frac{A_1 <: A_3 \hookrightarrow C}{A_1 \cap A_2 <: A_3 \hookrightarrow \lambda x{:}|A_1 \cap A_2|.\ C\ (\mathsf{proj}_1\, x)}\ \textsc{SubAnd}_1$$

$$\frac{A_2 <: A_3 \hookrightarrow C}{A_1 \cap A_2 <: A_3 \hookrightarrow \lambda x{:}|A_1 \cap A_2|.\ C\ (\mathsf{proj}_2\, x)}\ \textsc{SubAnd}_2$$

The crutial challenge lies in the generation of coercions that are derived by the subtyping rules. Since a program can typecheck via multiple derivations, and different derivation builds up multiple derivations, noncoherent arises.

If this situation occurs, we say that the semantics is *noncoherent*. More precisely, coherence is a property about the function that give meaning to valid programs. A system is coherent if any valid program has exactly one meaning.

Therefore, at this point two candidates of solutions occur:

- To forbid overlapping intersection types in a desired type system;
- To enforce an order of lookup. For example, the right item of a merge will take precedence so that it can "override" the left item.

With the second approach, the program above can only evaluate to 2. Unfortunately, although it is more liberal than the first, it makes equational reasoning broken in systems with parametric polymorphism.

Obviously the difficulty above is due to the fact that the type of $1,,2$, which is $\mathsf{Int} \cap \mathsf{Int}$ is an overlapping intersection. Generally, 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.

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 coercions.

2.4 Equational Reasoning

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

$$\mathsf{fst}\ \alpha\ \beta\ (x : \alpha \cap \beta) = (\lambda y{:}\alpha.\, y)\ x$$

What should be the result of this program?

```
fst Int Int (1,,2)
```

Then we have the following equational reasoning:

```
fst Int Int (1,,2) => (\(y : Int). y) (1,,2)
```

If we favour the second item, the program seems to evaluate to 2. But in reality, the result is 2. No matter we favour the first or the second item, we can always construct a program such that for that program, equational reasoning is broken.

Therefore, we require that the two types of an intersection must be not overlapping, or *disjoint*, and add this requirement to the well-formedness of types.

A well-formed type is such that given any query type, it is always clear which subpart the query is referring to. In terms of rules, this notion of well-formedness is almost the same as the one in System F except for intersection types we require the two components to be disjoint.

With parametric polymorphism, disjointness is harder to determine due to type variables. Consider this program:

$$\Lambda\alpha.\, \lambda x{:}\alpha \cap \mathsf{Int}.\, x$$

Types	A, B	$::=$	α
		$\mid$	$\bot$
		$\mid$	$A \to B$
		$\mid$	$\forall \alpha * B. A$
		$\mid$	$A \cap B$
Terms	e	$::=$	x
		$\mid$	$\lambda x{:}A. e$
		$\mid$	$e_1\ e_2$
		$\mid$	$\Lambda \alpha * A. e$
		$\mid$	$e\ A$
		$\mid$	$e_1 ,, e_2$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, \alpha * A \mid \Gamma, x{:}A$
Syntactic sugar	$\Lambda \alpha. e$	$\equiv$	$\Lambda \alpha * \bot. e$

Figure 1. Syntax.

x in the body is of type $\alpha \cap \mathsf{Int}$ and if α and Int are disjoint depends on the instantiation of α.

2.5 Intuition of Disjoint Quantification

Inspired by bounded quantification where a type variable is constrained by a type bound, we introduce the idea of disjoint quantification where a type variable is constrained to be disjoint with a given type.

There is a nice symmetry between bounded quantification and disjoint quantification. In systems with bounded quantification, the usual unconstrained quantifier $\forall \alpha. \ldots$ is a syntactic sugar for $\forall \alpha <: \top. \ldots$, and $\Lambda \alpha. \ldots$ for $\Lambda \alpha <: \top. \ldots$. In parellel, in our system with disjoint quantification, the usual unconstrained quantifier $\forall \alpha. \ldots$ is a syntactic sugar for $\forall \alpha * \bot. \ldots$, and $\Lambda \alpha. \ldots$ for $\Lambda \alpha * \top. \ldots$. The intuition is that since the bottom type is akin to the empty set, no other type overlaps with it.

With this tool in hand, we can rewrite the program above to:

$$\Lambda \alpha * \mathsf{Int}. \lambda x{:}\alpha \cap \mathsf{Int}. x$$

This program typechecks because while x is of type $\alpha \cap \mathsf{Int}$, and α is disjoint with Int. Similarly, in the new system, the original program no longer typechecks, thus preventing overlapping types.

3. The $F_{\&}$ calculus

This section presents the syntax, subtyping, and typing of $F_{\&}$, as well as the additional judgements that are special in $F_{\&}$. The semantics of $F_{\&}$ will be defined by a type-directed translation to a simple variant of System F in the next section.

3.1 Syntax

Figure 3.1 shows the syntax of $F_{\&}$ (with the addition to System F highlighted).

Meta-variables A, B range over types. Types include System F constructs: type variables α; function types $A \to B$; and type abstraction $\forall \alpha. A$. The bottom type $\bot$ is not inhabited by any term. $A \cap B$ denotes the intersection of types A and B. We omit type constants such as Int and String.

Terms include standard constructs in System F: variables x; abstraction of terms over variables of a given type $\lambda x : A. e$; application of terms to terms $e_1\ e_2$; and application of terms to types $e\ A$. "Big lambdas" $\Lambda \alpha * A. e$ abstracts a type variable α over a term e and constraints the instantiation of α to be disjoint with a given type A. $e_1 ,, e_2$ is the *merge* of two terms e_1 and e_2. It can be used as either e_1 or e_2. In particular, if one regards e_1 and e_2 as objects, their merge will respond to every method that one or both of them have. Merge of terms correspond to intersection types $A \cap B$.

In order to focus on the most essential features, we do not include other forms such as fixpoints here, although they are supported in our implementation and can be included in formalization in standard ways.

Typing contexts Γ track bound type variables with their disjointness constraint, and variables with their type A. We use use $[A/\alpha]\ B$ for the capture-avoiding substitution of A for α inside B and $\mathsf{ftv}(\cdot)$ for sets of free variables.

3.2 Subtyping

The subtyping rules of $F_{\&}$, shown in Figure 2, are syntax-directed (different from the approach by Davies and Pfenning [12], and Frisch et. al [18]). The rule (SUBFUN) says that a function is contravariant in its parameter type and covariant in its return type. A universal quantifier ($\forall$) is covariant in its body. The three rules dealing with intersection types are just what one would expect when interpreting types as sets. Under this interpretation, for example, the rule (SUBAND) says that if A_1 is both the subset of A_2 and the subset of A_3, then A_1 is also the subset of the intersection of A_2 and A_3. In order to achieve coherence, (SUBAND1) and (SUBAND2) additionally require the type on the right-hand side is atomic.

It is easy to see that subtyping is reflexive and transitive.

Lemma 1 (Subtyping is reflexive)**.** *Given a type* A, $A <: A$.

Lemma 2 (Subtyping is transitive)**.** *If* $A_1 <: A_2$ *and* $A_2 <: A_3$, *then* $A_1 <: A_3$.

For the corresponding mechanized proofs in Coq, we refer to the supplementary materials submitted with the paper.

3.3 Typing

The syntax-directed typing rules of $F_{\&}$ are shown in Figure 3. They consist of one main typing judgment and two auxiliary judgments. The main typing judgment is of the form: $\Gamma \vdash e : A$. It reads: "in the typing context Γ, the term e is of type A". The rules that are the same as in System F are rules for variables ((VAR)), lambda abstractions ((LAM)), and type applications ((TAPP)). For the ease of discussion, in (BLAM), we require the type variable introduced by the quantifier is fresh. For programs with type variable shadowing, this requirement can be met straighforwardly by variable renaming. The rule (APP) needs special attention as we add a subtyping requirement: the type of the argument (A_3) is a subtype of that of the parameter (A_1). For merges $e_1 ,, e_2$, we typecheck e_1 and e_2, check that the two resulting types are disjoint, and give it the intersection of the resulting types.

3.4 Type-directed translation to System F

In this section we define the dynamic semantics of the call-by-value $F_{\&}$ by means of a type-directed translation to a variant of System F. This translation turns merges into usual pairs, similar to Dunfield's elaboration approach [14]. In the end the translated terms can be typed and interpreted within System F. We add the blue-color part to our rules presented in the previous section. Besides that, they stay the same. We also tacitly assume the variables introduced in the blue part are generated from a unique name supply and are always fresh.

3.5 Informal discussion

This subsection presents the translation informally by explaining the major ideas.

Turning merges into pairs. The first idea is turning merges into pairs. For example,

$$1 ,, \texttt{"one"}$$

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

$$\frac{}{\alpha <: \alpha \hookrightarrow \lambda x{:}|\alpha|.\, x}\ \textsc{SubVar} \qquad \frac{A_3 <: A_1 \hookrightarrow C_1 \qquad A_2 <: A_4 \hookrightarrow C_2}{A_1 \to A_2 <: A_3 \to A_4 \hookrightarrow \lambda f{:}|A_1 \to A_2|.\, \lambda x{:}|A_3|.\, C_2\ (f\ (C_1\ x))}\ \textsc{SubFun}$$

$$\frac{A_1 <: [\alpha_1/\alpha_2]\, A_2 \hookrightarrow C}{\forall \alpha_1 * A_3.\, A_1 <: \forall \alpha_2 * A_3.\, A_2 \hookrightarrow \lambda f{:}|\forall \alpha_1 * A_3.\, A_1|.\, \Lambda \alpha.\, C\ (f\ \alpha)}\ \textsc{SubForall} \qquad \frac{A_1 <: A_2 \hookrightarrow C_1 \qquad A_1 <: A_3 \hookrightarrow C_2}{A_1 <: A_2 \cap A_3 \hookrightarrow \lambda x{:}|A_1|.\, (C_1\ x, C_2\ x)}\ \textsc{SubAnd}$$

$$\frac{A_1 <: A_3 \hookrightarrow C \qquad A_3 \text{ atomic}}{A_1 \cap A_2 <: A_3 \hookrightarrow \lambda x{:}|A_1 \cap A_2|.\, C\ (\mathsf{proj}_1 x)}\ \textsc{SubAnd}_1 \qquad \frac{A_2 <: A_3 \hookrightarrow C \qquad A_3 \text{ atomic}}{A_1 \cap A_2 <: A_3 \hookrightarrow \lambda x{:}|A_1 \cap A_2|.\, C\ (\mathsf{proj}_2 x)}\ \textsc{SubAnd}_2$$

2

Figure 2. Subtyping in $F_{\&}$.

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

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A \hookrightarrow x}\ \textsc{T\_Var} \qquad \frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x{:}A \vdash e : B \hookrightarrow E}{\Gamma \vdash \lambda x{:}A.\, e : A \to B \hookrightarrow \lambda x{:}|A|.\, E}\ \textsc{T\_Lam}$$

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

$$\frac{\Gamma, \alpha * B \vdash e : A \hookrightarrow E \qquad \Gamma \vdash B \text{ type} \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda \alpha * B.\, e : \forall \alpha * B.\, A \hookrightarrow \Lambda \alpha.\, E}\ \textsc{T\_BLam} \qquad \frac{\Gamma \vdash e : \forall \alpha * B.\, C \hookrightarrow E \qquad \Gamma \vdash A \text{ type} \qquad \Gamma \vdash A * B}{\Gamma \vdash e\ A : [A/\alpha]\, C \hookrightarrow E\ |A|}\ \textsc{T\_TApp}$$

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

Figure 3. The type system of $F_{\&}$.

becomes (1, "one"). In usage, the pair will be coerced according to type information. For example, consider the function application:

$$(\lambda x{:}\mathsf{String}.\, x)\ (1 ,, \text{"one"})$$

It will be translated to

$$(\lambda x{:}\mathsf{String}.\, x)\ ((\lambda x{:}(\mathsf{Int}, \mathsf{String}).\, \mathsf{proj}_2 x)\ (1, \text{"one"}))$$

The coercion in this case is $(\lambda x{:}(\mathsf{Int}, \mathsf{String}).\, \mathsf{proj}_2 x)$.
It extracts the second item from the pair since the function expects a String but the translated argument is of type $(\mathsf{Int}, \mathsf{String})$.

Erasing labels. The second idea is erasing record labels. For example,

$\{\mathsf{name} = \text{"Barbara"}\}$

becomes just "Barbara". To see how the this and the previous idea are used together, consider the following program:

$\{\mathsf{distance} = \{\mathsf{inKilometers} = 8,\ \mathsf{inMiles} = 5\}\}$

Since multi-field records are just merges, the record is desugared as

$\{\mathsf{distance} = \{\mathsf{inKilometers} = 8\} ,, \{\mathsf{inMiles} = 5\}\}$

and then translated to $(8,5)$.

Record operations as functions. The third idea is translating record operations into normal functions. For example, the source program

$\{\mathsf{distance} = \{\mathsf{inKilometers} = 8,\ \mathsf{inMiles} = 5\}\}.\mathsf{distance}.\,\mathsf{inMiles}$

becomes an $F_{\&}$ term

$$(\lambda x{:}(\mathsf{Int}, \mathsf{Int}).\, \mathsf{proj}_2 x)\ (8,5)$$

where $\lambda x{:}(\mathsf{Int}, \mathsf{Int}).\, \mathsf{proj}_2 x$ extracts the desired item 5.

3.6 Target language

Our target language is System F extended with pair and unit types. The syntax and typing is completely standard. The syntax of the target language is shown in Figure 4 and the typing rules in the appendix.

Types	T	$::=$	$\alpha \mid () \mid T_1 \to T_2 \mid \forall \alpha.\, T \mid (T_1, T_2)$
Terms	E, C	$::=$	$x \mid () \mid \lambda x{:}T.\, E \mid E_1\ E_2 \mid \Lambda \alpha.\, E$
		$\mid$	$E\ T \mid (E_1, E_2) \mid \mathsf{proj}_k E$
Contexts	Γ	$::=$	$\epsilon \mid \Gamma, \alpha \mid \Gamma, x{:}T$

Figure 4. Target language syntax.

3.7 Type translation

Figure 5 defines the type translation function $|\cdot|$ from $F_{\&}$ types A to target language types T. The notation $|\cdot|$ is also overloaded for context translation from $F_{\&}$ contexts γ to target language contexts Γ.

3.8 Coercive subtyping

Figure shows subtyping with coercions. The judgment

fig:elab-subtyping

$$A_1 <: A_2 \hookrightarrow C$$

$\boxed{|A| = T}$

$$
\begin{aligned}
|\alpha| &= \alpha \\
|\top| &= () \\
|A_1| \to |A_2| &= |A_1| \to |A_2| \\
|\forall \alpha.\, A| &= \forall \alpha.\, |A| \\
|A_1 \cap A_2| &= (|A_1|, |A_2|)
\end{aligned}
$$

$\boxed{|\gamma| = \Gamma}$

$$
\begin{aligned}
|\epsilon| &= \epsilon \\
|\gamma, \alpha| &= |\gamma|, \alpha \\
|\gamma, \alpha{:}A| &= |\gamma|, \alpha{:}|A|
\end{aligned}
$$

Figure 5. Type and context translation.

extends the subtyping judgment in Figure 2 with a coercion on the right hand side of $\hookrightarrow$. A coercion C is just an term in the target language and is ensured to have type $|A_1| \to |A_2|$ (Lemma 3)BRUNO: ref now showing. For example,

$$\mathsf{Int} \cap \mathsf{Bool} <: \mathsf{Bool} \hookrightarrow \lambda x{:}|\mathsf{Int} \cap \mathsf{Bool}|.\, \mathsf{proj}_2\, x$$

generates a coercion function from $\mathsf{Int} \cap \mathsf{Bool}$ to Bool.

In rules (SUBVAR), (SUBTOP), (SUBFORALL), coercions are just identity functions. In (SUBFUN), we elaborate the subtyping of parameter and return types by η-expanding f to $\lambda x{:}|A_3|.\, f\, x$, applying C_1 to the argument and C_2 to the result. Rules (SUBAND1), (SUBAND2), and (SUBAND) elaborate with intersection types. (SUBAND) uses both coercions to form a pair. Rules (SUBAND1) and (SUBAND2) reuse the coercion from the premises and create new ones that cater to the changes of the argument type in the conclusions. Note that the two rules are syntatically the same and hence a program can be elaborated differently, depending on which rule is used. But in the implementation one usually applies the rules sequentially with pattern matching, essentially defining a deterministic order of lookup.

Lemma 3 (Subtyping rules produce type-correct coercion)**.** *If* $A_1 <: A_2 \hookrightarrow C$*, then* $e \vdash C : |A_1| \to |A_2|$.

Proof. By a straighforward induction on the derivation[3]. □

3.9 Main translation

Main translation judgment. The main translation judgment $\gamma \vdash e : A \hookrightarrow E$ extends the typing judgment with an elaborated term on the right hand side of $\hookrightarrow$. The translation ensures that E has type $|A|$. In $F_{\&}$, one may pass more information to a function than what is required; but not in System F. To account for this difference, in (APP), the coercion C from the subtyping relation is applied to the argument. (MERGE) straighforwardly translates merges into pairs.

Theorem 1 (Translation preserves well-typing)**.** *If* $\gamma \vdash e : A \hookrightarrow E$*, then* $|\gamma| \vdash E : |A|$.

Proof. (Sketch) By structural induction on the term and the corresponding inference rule. □

Theorem 2 (Type safety)**.** *If* e *is a well-typed* $F_{\&}$ *term, then* e *evaluates to some System* F *value* v.

[3] The proofs of major lemmata and theorems can be found in the appendix.

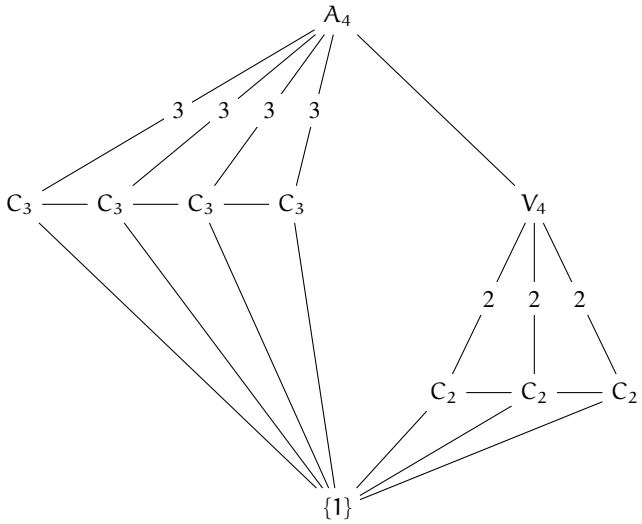

Figure 6. Untergruppenverband

Proof. Since we define the dynamic semantics of $F_{\&}$ in terms of the composition of the type-directed translation and the dynamic semantics of System F, type safety follows immediately. □

4. Disjoint intersection types and disjoint quantification

This section shows how to restrict the system presented before so that it supports coherence as well as type soundness. The keys aspects are the notion of disjoint intersections, and disjoint quantification for polymorphic types.

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.

The intuition can be found in figure ....

Restrictions on subtyping :

The subtyping rules, without the atomic condition are overlapping. With the atomic constraint, one can guarantee that at any moment during the derivation of a subtyping relation, at most one rule can be used. Indeed, our restrictions on subtyping do not make the subtyping relation less expressive to one without such restrictions.

Point to proofs

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 $A_1 \cap A_2 <: A_3$, we would like to be certain that either $A_1 <: A_3$ holds or $A_2 <: A_3$ holds, but not both.

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 A_1 and A_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 $\mathsf{Nat} <:_{\mathsf{prim}} \mathsf{Int}$ (a natural number is also an integer), which can be promoted to the normal subtyping with this rule:

$$\frac{A_1 <:_{\mathsf{prim}} A_2}{A_1 <: A_2}$$

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.

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.

GEORGE: Add interpretation of the theorem

Theorem 3. *If* $A <: C$*, then* $A \cap B <: C$*. If* $B <: C$*, then* $A \cap B <: C$*.*

Proof. By induction on C. If $C \neq C_1 \cap C_2$, trivial. If $C = C_1 \cap C_2$, Need to show $A <: C_1 \cap C_2$ implies $A \cap B <: C_1 \cap C_2$. By inversion $A <: C_1$ and $A <: C_2$. By the i.h., $A \cap B <: C_1$ and $A \cap B <: C_2$. By (SUBAND), $A \cap B <: C_1 \cap C_2$. □

4.1 Disjointness

Spec of disjointness/intuition ...

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

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

4.2 Well-formed types

4.3 Subtyping

4.4 Metatheory

Definition 2. Type variable constraint We say the *constraint* of a type variable α inside the context Γ is A if $\alpha * A \in \Gamma$.

Lemma 4 (Free type variables of disjoint bounds)**.** *If* $\Gamma \vdash \alpha * A$*, then* $\alpha \notin ftv(A)$*.*

Lemma 5 (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 6 (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 $B = B_1 \cap B_2$[4]: Since B is not atomic, only (SUBAND) apply.

□

In general, disjointness judgements are not invariant with respect to free-variable substitution. In other words, a careless substitution can violate the disjoint constraint in the context. For example, in the context $\alpha * \mathsf{Int}$, α and Int are disjoint:

$$\alpha * \mathsf{Int} \vdash \alpha * \mathsf{Int}$$

But after the substitution of Int for α on the two types, the sentence

$$\alpha * \mathsf{Int} \vdash \mathsf{Int} * \mathsf{Int}$$

is longer true since Int is clearly not disjoint with itself.

Lemma 7. *Invariance of disjointness If* $\Gamma \vdash A * B$ *and* R *respects the constraints of* β*, then* $\Gamma \vdash [R/\beta]\, A * [R/\beta]\, B$.

This lemma says that substitution for free type variables preserves disjointness of types if the combination of the replacement type and the type variable is proven disjoint.

Proof. By induction on the derivation of $\Gamma \vdash A * B$.

- Case
$$\frac{\alpha * B \in \Gamma}{\Gamma \vdash \alpha *_I B}\ \textsc{DisjointVar}$$
We need to show
$$\Gamma \vdash [R/\beta]\ \alpha * [R/\beta]\ B$$
If β is not equivalent to α and is not free in B, then the above trivially holds by the def. of the substitution function. Otherwise, if β is equivalent to α, then we need to show
$$\Gamma \vdash R * [R/\beta]\ B$$
- Case
$$\frac{\Gamma \vdash A *_I C \qquad \Gamma \vdash B *_I C}{\Gamma \vdash A \cap B *_I C}\ \textsc{DisjointInter1}$$
By applying the i.h. and the def. of the substitution function.
- Case
$$\frac{\Gamma \vdash A *_I B \qquad \Gamma \vdash A *_I C}{\Gamma \vdash A *_I B \cap C}\ \textsc{DisjointInter2}$$
Similar.
- Case
$$\frac{\Gamma \vdash B *_I D}{\Gamma \vdash A \to B *_I C \to D}\ \textsc{DisjointFun}$$
By applying the i.h. and the def. of the substitution function.
- Case
$$\frac{\Gamma \vdash A *_I C}{\Gamma \vdash \forall \alpha * B.\, A *_I \forall \alpha * B.\, \text{CSubst???}}\ \textsc{DisjointForall}$$
By applying the i.h. and the def. of the substitution function. Note that α is fresh.

[4] An example of this case is:

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

- Case
$$\frac{A \not\sim B}{\Gamma \vdash A *_I B}\ \textsc{DisjointAtomic}$$
Substitution does not change the shape of types when the variable case is excluded. Therefore, the relation in the premise of the rule continue to hold and hence the conclusion.

□

Lemma 8. *Substitution If* $\Gamma \vdash R$ *type,* $\Gamma \vdash S$ *type, and* R *respects the constraints of* β*, then* $\Gamma \vdash [R/\beta]\ S$ *type.*

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

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

Subst. of A
- Case
$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type} \qquad \Gamma \vdash A * B}{\Gamma \vdash A \cap B \text{ type}}\ \textsc{WFInter}$$
By i.h., $\Gamma \vdash [R/\beta]\ A$ type and $\Gamma \vdash [R/\beta]\ B$ type. By Lemma 7, we also have $\Gamma \vdash [R/\beta]\ A * [R/\beta]\ B$. Therefore by (WFInter), $\Gamma \vdash [R/\beta]\ A \cap B$ type.

□

Lemma 9. *Instantiation If* $\Gamma, \alpha * B \vdash C$ *type,* $\Gamma \vdash A$ *type,* $\Gamma \vdash A * B$ *then* $\Gamma \vdash [A/\alpha]\ C$ *type.*

Proof. By induction.

- Case
$$\frac{\alpha * A \in \Gamma}{\Gamma \vdash \alpha \text{ type}}\ \textsc{WFVar}$$
If $C = \alpha$, then $[A/\alpha]\ \alpha = A$. Since $\Gamma \vdash A$ type, it follows that $\Gamma \vdash [A/\alpha]\ \alpha$ type; otherwise, let $C = \beta$, where β is a type variable distinct from α. Since $\Gamma, \alpha * B \vdash \beta$ type and α and β are distinct, β must be in Γ and therefore $\Gamma \vdash \beta$ type, which is equivalent to $\Gamma \vdash [A/\alpha]\ \beta$ type.
- Case
$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \to B \text{ type}}\ \textsc{WFFun}$$
By straightforwardly applying the i.h and the rule itself.
- Case
$$\frac{}{\Gamma \vdash \bot \text{ type}}\ \textsc{WFBot}$$
Trivial.
- Case
$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, \alpha * A \vdash B \text{ type}}{\Gamma \vdash \forall \alpha * A.\ B \text{ type}}\ \textsc{WFForall}$$
By straightforwardly applying the i.h and the rule itself.
- Case
$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type} \qquad \Gamma \vdash A * B}{\Gamma \vdash A \cap B \text{ type}}\ \textsc{WFInter}$$
Let C in the statement of this lemma be $C_1 \cap C_2$. By the condition we know
$$\Gamma, \alpha * B \vdash C_1 \cap C_2 \text{ type}$$
Thus we must have,
$$\Gamma, \alpha * B \vdash C_1 \text{ type}$$
By the i.h., $\Gamma \vdash [A/\alpha]\ C_1$ type and similarly $\Gamma \vdash [A/\alpha]\ C_2$ type. By (WFInter),
$$\Gamma \vdash [A/\alpha]\ C_1 \cap [A/\alpha]\ C_2 \text{ type}$$
and hence
$$\Gamma \vdash [A/\alpha]\ (C_1 \cap C_2) \text{ type}$$

Show disjointness

□

Lemma 10. *Well-formed typing If* $\Gamma \vdash e : A$*, then* $\Gamma \vdash e$ *type.*

Typing always produces a well-formed type.

Proof. By induction on the derivation of $\Gamma \vdash e : A$. The case of (TyTApp) needs special attention
$$\frac{\Gamma \vdash e : \forall \alpha * B.\ C \hookrightarrow E \qquad \Gamma \vdash A \text{ type} \qquad \Gamma \vdash A * B}{\Gamma \vdash e\ A : [A/\alpha]\ C \hookrightarrow E\ |A|}\ \text{T\_TApp}$$
because we need to show that the result of substitution $([A/\alpha]\ C)$ is well-formed, which is evident by Lemma 9. □

Theorem 4 (Unique elaboration)**.** *If* $\Gamma \vdash e : A_1 \hookrightarrow E_1$ *and* $\Gamma \vdash e : A_2 \hookrightarrow E_2$*, then* $E_1 \equiv E_2$.

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

Proof. The typing rules are syntax-directed. The case of (TyApp) needs special attention since we still need to show that the generated coercion C is unique.
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A_1 \to A_2 \hookrightarrow E_1 \\ \Gamma \vdash e_2 : A_3 \hookrightarrow E_2 \qquad A_3 <: A_1 \hookrightarrow C\end{array}}{\Gamma \vdash e_1\ e_2 : A_2 \hookrightarrow E_1\ (C\ E_2)}\ \text{T\_App}$$
By Lemma 10, we have $\Gamma \vdash A_1$ type and $\Gamma \vdash A_3$ type. Therefore we are able to apply Lemma 6 and conclude that C is unique. □

4.5 Disjointness and well-formedness

The well-formedness of types is standard except that the two components of an intersection type must be disjoint.

$\boxed{A \text{ atomic}}$

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

Figure 7. Atomic types.

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

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

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

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

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type} \qquad \Gamma \vdash A * B}{\Gamma \vdash A \cap B \text{ type}}\ \textsc{WFInter}$$

Figure 8. Well-formed types.

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

$$\frac{\alpha * B \in \Gamma}{\Gamma \vdash \alpha *_I B}\ \textsc{DisjointVar}$$

$$\frac{\Gamma \vdash A *_I C \qquad \Gamma \vdash B *_I C}{\Gamma \vdash A \cap B *_I C}\ \textsc{DisjointInter1}$$

$$\frac{\Gamma \vdash A *_I B \qquad \Gamma \vdash A *_I C}{\Gamma \vdash A *_I B \cap C}\ \textsc{DisjointInter2}$$

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

$$\frac{\Gamma \vdash A *_I C}{\Gamma \vdash \forall \alpha * B.\, A *_I \forall \alpha * B.\, \text{CSubst???}}\ \textsc{DisjointForall}$$

$$\frac{A \not\sim B}{\Gamma \vdash A *_I B}\ \textsc{DisjointAtomic} \qquad \boxed{A \not\sim B}$$

$\bot \not\sim A \to B$ NotSimBot1 $\qquad \bot \not\sim \forall \alpha * B.\, A$ NotSimBot2

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

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

Figure 9. Algorithmic disjointness.

5. Algorithmic disjointness

5.1 Motivation

The rules for the disjointness judgement are shown in Figure 5.1. The judgement says two types A and B are disjoint in a context Γ. Two atomic types with different shapes (except for the variable) are considered disjoint, which is factored out to the atomic disjointness rules. The (DisjointInter1) and (DisjointInter2) inductively distribute the relation itself over the intersection constructor ($\cap$). (DisjointFun) is quite interesting, because it says two function types are disjoint as long as their return types are disjoint (regardless of their parameter types).

Although the system in the previous section shows a formal system of disjoint intersection types, it relies on a non-algorithmic specification of disjointness. This section shows an algorithmic specification of disjointness that is proved to be sound and complete.

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 A 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.)

$$\Uparrow(\alpha, \alpha) = \{\alpha\}$$
$$\Uparrow(\bot, \bot) = \{\bot\}$$
$$\Uparrow(A_1 \to A_2, A_3 \to A_4) = \Downarrow(A_1, A_3) \to \Uparrow(A_2, A_4)$$

Notation. We use $\Downarrow(A_1, A_3) \to \Uparrow(A_2, A_4)$ as a shorthand for $\{s \to t \mid s \in \Downarrow(A_1 \to A_2), t \in \Uparrow(A_2, A_4)\}$. Therefore, the problem of determining if $\Downarrow(A_1, A_3) \to \Uparrow(A_2, A_4)$ is empty reduces to the problem of determining if $\Uparrow(A_2, A_4)$ is empty.

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

5.2 Formal system

Explain the rules and intuitions.

The algorithmic rules for disjointness is sound and complete.

Lemma 11. *Symmetry of disjointness If* $\Gamma \vdash A * B$*, then* $\Gamma \vdash B * A$.

Proof. Trivial by the definition of disjointness. □

Theorem 5. *If* $\Gamma \vdash A * C$ *and* $\Gamma \vdash B * C$*, then* $\Gamma \vdash A \cap B * C$.

Lemma 12. *If* $A_1 \to A_2 <: D$ *and* $B_1 \to B_2 <: D$*, then there exists a* C *such that* $A_2 <: C$ *and* $B_2 <: C$.

Proof. By induction on D. □

Theorem 6. *Soundness For any two types* A, B, $\Gamma \vdash A *_I B$ *implies* $\Gamma \vdash A * B$.

Proof. By induction on $*_I$.

- Case
$$\frac{\Gamma \vdash B *_I D}{\Gamma \vdash A \to B *_I C \to D}\ \textsc{DisjointFun}$$
Lemma 12
GEORGE: May need an extracted lemma here
- Case
$$\frac{\Gamma \vdash A *_I C \qquad \Gamma \vdash B *_I C}{\Gamma \vdash A \cap B *_I C}\ \textsc{DisjointInter1}$$
By Lemma 5 and the i.h.
- Case
$$\frac{\Gamma \vdash A *_I B \qquad \Gamma \vdash A *_I C}{\Gamma \vdash A *_I B \cap C}\ \textsc{DisjointInter2}$$
By Lemma 5, Lemma 11, and the i.h.
- Case
$$\frac{A \not\sim B}{\Gamma \vdash A *_I B}\ \textsc{DisjointAtomic}$$
Need to show ... By unfolding the definition of disjointness Need to show there does not exists C such that... By induction on C. Atomic cases... If $C = C_1 \cap C_2$ By inversion and the i.h. we arrive at a contradiction.

□

Theorem 7. *Completeness For any two type* A, B, $\Gamma \vdash A * B$ *implies* $\Gamma \vdash A *_I B$.

Proof. Induction on A.

- Case $\bot$
Induction on B.
  - Case $B = \bot$
Need to show $\Gamma \vdash \bot * \bot$ implies $\Gamma \vdash \bot *_I \bot$. Take $C = \bot$. Clearly the premise is false by definition. Then the whole statement is true.
  - Case $B = B_1 \to B_2$ The conclusion is true by the disjoint axioms.
  - Case $B = B_1 \cap B_2$. Need to show $\Gamma \vdash \bot * B_1 \cap B_2$ implies $\Gamma \vdash \bot *_I B_1 \cap B_2$. Apply (DisjointInter2) and the resulting conditions can be proved by the i.h.
  - Case
- $A = A_1 \to A_2$
  - Case $B = \bot$ The conclusion is true by the disjoint axioms.
  - Case $B = B_1 \to B_2$ Need to show $\Gamma \vdash A_1 \to A_2 * B_1 \to B_2$ implies $\Gamma \vdash A_1 \to A_2 *_1 B_1 \to B_2$. Apply (DisjointFun) and the result, $\Gamma \vdash A_2 *_1 B_2$, can be proved by the i.h.
  - Case $B = B_1 \cap B_2$. Need to show $\Gamma \vdash A_1 \to A_2 * B_1 \cap B_2$ implies $\Gamma \vdash A_1 \to A_2 *_I B_1 \cap B_2$. Apply (DisjointInter2) and the resulting conditions can be proved by the i.h.
- $A = A_1 \cap A_2$ By (DisjointInter1) and by the i.h.

???

□

6. Implementation

We implemented the core functionalities of the $F_{\&}$ as part of a JVM-based compiler. The implementation supports record update instead of restriction as a primitive; however the former is formalized with the same underlying idea of elaborating records. Based on the type system of $F_{\&}$, we built an ML-like source language compiler that offers interoperability with Java (such as object creation and method calls). The source language is loosely based on the more general System F_ω and supports a number of other features, including multi-field records, mutually recursive `let` bindings, type aliases, algebraic data types, pattern matching, and first-class modules that are encoded with `letrec` and records.

Relevant to this paper are the three phases in the compiler, which collectively turn source programs into System F:

1. A *typechecking* phase that checks the usage of $F_{\&}$ features and other source language features against an abstract syntax tree that follows the source syntax.
2. A *desugaring* phase that translates well-typed source terms into $F_{\&}$ terms. Source-level features such as multi-field records, type aliases are removed at this phase. The resulting program is just an $F_{\&}$ term extended with some other constructs necessary for code generation.
3. A *translation* phase that turns well-typed $F_{\&}$ terms into System F ones.

Phase 3 is what we have formalized in this paper.

Removing identity functions. Our translation inserts identity functions whenever subtyping or record operation occurs, which could mean notable run-time overhead. But in practice this is not an issue. In the current implementation, we introduced a partial evaluator with three simple rewriting rules to eliminate the redundant identity functions as another compiler phase after the translation. In another version of our implementation, partial evaluation is weaved into the process of translation so that the unwanted identity functions are not introduced during the translation.

7. Related work

Intersection types with polymorphism. Our type system combines intersection types and parametric polymorphism. Closest to us is Pierce's work [28] on a prototype compiler for a language with both intersection types, union types, and parametric polymorphism. Similarly to $F_{\&}$ in his system universal quantifiers do not support bounded quantification. However Pierce did not try to prove any meta-theoretical results and his calculus does not have a merge operator. Pierce also studied a system where both intersection types and bounded polymorphism are present in his Ph.D. dissertation [29] and a 1997 report [30]. Going in the direction of higher kinds, Compagnoni and Pierce [7] added intersection types to System F_ω and used the new calculus, $F^\omega_\wedge$, to model multiple inheritance. In their system, types include the construct of intersection of types of the same kind K. Davies and Pfenning [12] studied the interactions between intersection types and effects in call-by-value languages. And they proposed a "value restriction" for intersection types, similar to value restriction on parametric polymorphism. Although they proposed a system with parametric polymorphism, their subtyping rules are significantly different from ours, since they consider parametric polymorphism as the "infinit analog" of intersection polymorphism. There have been attempts to provide a foundational calculus for Scala that incorporates intersection types [1, 2]. Although the minimal Scala-like calculus does not natively support parametric polymorphism, it is possible to encode parametric polymorphism with abstract type members. Thus it can be argued that this calculus also supports intersection types and parametric polymorphism. However, the type-soundness of a minimal Scala-like calculus with intersection types and parametric polymorphism is not yet proven. Recently, some form of intersection types has been adopted in object-oriented languages such as Scala, Ceylon, and Grace. Generally speaking, the most significant difference to $F_{\&}$ is that in all previous systems there is no explicit introduction construct like our merge operator. As shown in Section **??**, this feature is pivotal in supporting modularity and extensibility because it allows dynamic composition of values.

Other type systems with intersection types. Intersection types date back to as early as Coppo et al. [9]. As emphasized throughout the paper our work is inspired by Dunfield [14]. He described a similar approach to ours: compiling a system with intersection types into ordinary λ-calculus terms. The major difference is that his system does not include parametric polymorphism, while ours does not include unions. Besides, our rules are algorithmic and we formalize a record system. Reynolds invented Forsythe [34] in the 1980s. Our merge operator is analogous to his p_1, p_2. As Dunfield has noted, in Forsythe merges can be only used unambiguously. For instance, it is not allowed in Forsythe to merge two functions.

Refinement intersection [11, 13, 17] is the more conservative approach of adopting intersection types. It increases only the expressiveness of types but not terms. But without a term-level construct like "merge", it is not possible to encode various language features. As an alternative to syntatic subtyping described in this paper, Frisch et al. [18] studied semantic subtyping.

Languages for extensibility. To improve support for extensibility various researchers have proposed new OOP languages or programming mechanisms. It is interesting to note that design patterns such as object algebras or modular visitors provide a considerably different approach to extensibility when compared to some previous proposals for language designs for extensibility. Therefore the requirements in terms of type system features are quite different. One popular approach is *family polymorphism* [15], which allows whole class hierarchies to be captured as a family of classes. Such a family can be later reused to create a derived family with potentially new class members, and additional methods in the existing classes. *Virtual classes* [16] are a concrete realization of this idea, where a container class can hold nested inner *virtual* classes (forming the family of classes). In a subclass of the container class, the inner classes can themselfves be *overriden*, which is why they are called virtual. There are many language mechanisms that provide variants of virtual classes or similar mechanisms [3, 22, 24, 35]. The work by Nystrom on *nested intersection* [24] uses a form of intersection types to support the composition of families of classes. Ostermann's *delegation layers* [27] use delegation for doing dynamic composition in a system with virtual classes. This in contrast with most other approaches that use class-based composition, but closer to the dynamic composition that we use in $F_{\&}$.

8. Conclusion and further work

We have described a simple type system suitable for extensible designs. The system has a term-level introduction form for intersection types, combines intersection types with parametric polymorphism, and supports extensible records using a lightweight mechanism. We prove that the translation is type-preserving and the language is type-safe.

There are various avenues for future work. On the one hand we are interested in creating a source language where extensible designs such as object algebras or modular visitors are supported by proper language features. On the other hand we would like to explore extending our structural type system with nominal subtyping to allow more familiar programming experience.

References

[1] N. Amin, A. Moors, and M. Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, 2012.

[2] N. Amin, T. Rompf, and M. Odersky. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Transactions on aspect-oriented software development i. chapter An Overview of Caesarj. 2006.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, 1998.

[5] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, 1989.

[6] L. Cardelli, S. Martini, J. Mitchell, and A. Scedrov. An extension of System F with subtyping. *Information and Computation*, 1994.

[7] A. B. Compagnoni and B. C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 1996.

[8] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1989.

[9] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 1981.

[10] B. C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In *23rd European Conference on Object Oriented Programming (ECOOP)*, 2009.

[11] R. Davies. *Practical refinement-type checking*. PhD thesis, University of Western Australia, 2005.

[12] R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, 2000.

[13] J. Dunfield. Refined typechecking with stardust. In *Proceedings of the 2007 workshop on Programming languages meets program verification*. ACM, 2007.

[14] J. Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 2014.

[15] E. Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, 2001.

[16] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. *POPL 2006*.

[17] T. Freeman and F. Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, 1991.

[18] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)*, 2008.

[19] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, 1993.

[20] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, (5), 2006.

[21] D. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 1968.

[22] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fasioned java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, 2001.

[23] A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, 2008.

[24] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested intersection for scalable software composition. In *In Proc. 2006 OOPSLA*.

[25] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses. In *ECOOP 2012–Object-Oriented Programming*. 2012.

[26] B. C. d. S. Oliveira, T. Van Der Storm, A. Loh, and W. R. Cook. Feature-oriented programming with object algebras. In *ECOOP 2013–Object-Oriented Programming*. 2013.

[27] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, 2002.

[28] B. C. Pierce. Programming with intersection types, union types, and polymorphism. 1991.

[29] B. C. Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 1991.

[30] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 1997.

[31] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP '97 — Object-Oriented Programming 11th European Conference*. 1997.

[32] T. Rendel, J. I. Brachthäuser, and K. Ostermann. From object algebras to attribute grammars. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, 2014.

[33] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, 1974.

[34] J. C. Reynolds. *Design of the programming language Forsythe*. 1997.

[35] Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, 1998.

[36] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, 1999.

[37] M. Torgersen. The Expression Problem Revisited. In M. Odersky, editor, *Proc. of the 18th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, 2004.

[38] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the java programming language. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, 2004.

[39] P. Wadler. The expression problem. *Java-genericity mailing list*, 1998.

[40] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *FOOL*, 2005.

A. Type well-formedness

B. Target Type System

$\boxed{\Gamma \vdash E : T}$

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T}\ \textbf{T}_{\text{VAR}} \qquad \frac{}{\Gamma \vdash () : ()}\ \textbf{T}_{\text{UNIT}} \qquad \frac{\Gamma, x{:}T \vdash E : T_1 \qquad \Gamma \vdash T}{\Gamma \vdash \lambda x{:}T.\, E : T \to T_1}\ \textbf{T}_{\text{LAM}} \qquad \frac{\Gamma \vdash E_1 : T_1 \to T_2 \qquad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1\ E_2 : T_2}\ \textbf{T}_{\text{APP}}$$

$$\frac{\Gamma, \alpha \vdash E : T}{\Gamma \vdash \Lambda\alpha.\, E : \forall\alpha.\, T}\ \textbf{T}_{\text{BLAM}} \qquad \frac{\Gamma \vdash E : \forall\alpha.\, T_1 \qquad \Gamma \vdash T}{\Gamma \vdash E\ T : [T/\alpha]\ T_1}\ \textbf{T}_{\text{TAPP}} \qquad \frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : (T_1, T_2)}\ \textbf{T}_{\text{PAIR}} \qquad \frac{\Gamma \vdash E : (T_1, T_2)}{\Gamma \vdash \mathsf{proj}_1 E : T_1}\ \textbf{T}_{\text{PROJ}_1}$$

$$\frac{\Gamma \vdash E : (T_1, T_2)}{\Gamma \vdash \mathsf{proj}_2 E : T_2}\ \textbf{T}_{\text{PROJ}_2}$$

Figure 10. Target type system.