Disjoint Intersection Types and Disjoint Quantification

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

Previous work by Dunfield has shown that a simply typed core calculus with intersection types and a merge operator provides a powerful foundation for various programming language features. However, while his calculus is type-safe, it lacks the important property of *coherence*: different derivations for the same expression can lead to different results. The lack of coherence is important disavantage for adoption of his core calculus in implementations of programming languages, as the semantics of the programming language becomes implementation dependent.

This paper presents $F_{\&}$: a core calculus with a variant of *intersection types*, *parametric polymorphism* and a *merge operator*. The semantics $F_{\&}$ is both type-safe and coherent. Coherence is achieved by ensuring that intersection types are *disjoint*. Formally two types are disjoint if they do not share a common supertype. We present a type system that prevents intersection types that are not disjoint, as well as an algorithmic specification to determine whether two types are disjoint. Moreover we show that this approach extends to systems with parametric polymorphism. Parametric polymorphism makes the problem of coherence significantly harder. When a type variable occurs in an intersection type, it is not statically known whether the instantiated type will share a common supertype with other components of the intersection. To address this problem we propose *disjoint quantification*: a constrained form of parametric polymorphism, that allows programmers to specify disjointness constraints for type variables. With disjoint quantification the calculus remains very flexible in terms of programs that can be written with intersection types, while retaining coherence.

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

General Terms Design, Languages, Theory

Keywords Intersecion Types, Polymorphism, Type System

1. Introduction

Previous work by Dunfield [] has shown the power of intersection types and a merge operator. The presence of a merge operator in a core calculus provides significant expressiveness, allowing encodings for many other language constructs as syntactic sugar. For example single-field records are easily encoded as types with a label, and multi-field records are encoded as the concatenation of single-field records. Concatenation of records is expressed using intersection types at the type-level and the corresponding merge operator and the term level. Dunfield formalized a simply typed lambda calculus with intersection types and the merge operator, and showed how to give a semantics to the calculus by a type-directed translation to a simply typed lambda calculus extended with pairs. The type-directed translation is simple, elegant, and type-safe.

Intersection types and the merge operator are also useful in the context of software *extensibility*. In recent years there has been a wide interest in presenting solutions to the *expression problem* [] in various communities. Currently there are various solutions in the functional programming, object-oriented programming and theorem proving communities. Morever, intersection types and an *encoding* of a merge operator have been shown to be useful to solve additional challenges related to extensibility []. Many solutions follow the similar ideas at their core. In particular, various solutions are closely related to type-theoretic encodings of datatypes []. However the use of language-specific mechanisms to express those solutions hides their essence. Clearly this seems to indicate that a more foundational approach is lacking. What is currently missing is a foundational core calculus that can capture the key ideas behind the various solutions. We believe that such calculus should support parametric polymorphism, intersection types and a merge operator.

Dunfield calculus seems to provide a good basis for a foundational calculus for studying extensibility. However, his calculus is still insufficient for extensibility in two accounts. Firstly it does not support parametric polymorhism. This is a pressing limitation because type-theoretic encodings of datatypes fundamentally rely on parametric polymorphism. Secondly, and more importantly, while Dunfield calculus is type-safe, it lacks the property of *coherence*: different derivations for the same expression can lead to different results. The lack of coherence is important disavantage for adoption of his core calculus in implementations of programming languages, as the semantics of the programming language becomes implementation dependent. BRUNO: Moreover, from a theoretical point of view the lack of coherence also makes the semantics "imprecise", which is unfortunate.

This paper presents $F_{\&}$: a core calculus with a variant of *intersection types*, *parametric polymorphism* and a *merge operator*. The semantics $F_{\&}$ is both type-safe and coherent. Thus $F_{\&}$ addresses the two limitations of Dunfield calculus and can be used to express the key ideas of extensible type-theoretic encodings of datatypes.

Coherence is achieved by ensuring that intersection types are *disjoint*. Given two types A and B, two types are disjoint ($A * B$) if there is no type C such that both A and B are subtypes of C. Formally this definition is captured as follows:

$$A * B \equiv \nexists C.\ A <: C \wedge B <: C$$

With this definition of disjointness we present a formal specification of a type system that prevents intersection types that are not disjoint. However, the mathematical definition of disjointness is not readely implementable. Therefore, we also present an algorithmic specification to determine whether two types are disjoint, and we show that this algorithmic specification is sound and complete with respect to the mathmatical definition of disjointness.

We also show that disjoint intersection types can be extended to support parametric polymorphism. Parametric polymorphism makes the problem of coherence significantly harder. When a type variable occurs in an intersection type, it is not statically known whether the instantiated type will share a common supertype with other components of the intersection. To address this problem we

propose *disjoint quantification*: a constrained form of parametric polymorphism, that allows programmers to specify disjointness constraints for type variables. With disjoint quantification the calculus remains very flexible in terms of programs that can be written with intersection types, while retaining coherence.

In summary, the contributions of this paper are:

- **Disjoint Intersection Types:** A new form of intersection type where only disjoint types (types that do not share a common super type) are allowed. A sound and complete algorithmic specification for determining whether two types are disjoint is presented.
- **Disjoint Quantification:**: A novel form of universal quantification where type variables can have disjointness constraints.
- **Formalization of System $F_{\&}$ and Proof of Coherence:** An elaboration semantics of System $F_{\&}$ into System F is given. Type-soundness and coherence are proved.
- **Encodings of Extensible Designs:** Various encodings of extensible designs into System $F_{\&}$, including *Object Algebras* and *Modular Visitors*.
- **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

A number of OO languages, such as Java, C#, Scala, and Ceylon[2], already support intersection types to different degrees. In Java, for example,

```
interface AwithB extends A, B {}
```

introduces a new interface AwithB that satisfies the interfaces of both A and B. Arguably such type can be considered as a nominal intersection type. Scala takes one step further by eliminating the need of a nominal type. For example, given two concrete traits, it is possible to use *mixin composition* to create an object that implements both traits. Such an object has a (structural) intersection type:

```
trait A
trait B

val newAB : A with B = new A with B
```

Scala also allows intersection of type parameters. For example:

```
def merge[A,B] (x: A) (y: B) : A with B = ...
```

uses the annonymous intersection of two type parameters A and B. However, in Scala it is not possible to dynamically compose two objects. For example, the following code:

```
// Invalid Scala code:
def merge[A,B] (x: A) (y: B) : A with B = x with y
```

is rejected by the Scala compiler. The problem is that the with construct for Scala expressions can only be used to mixin traits or classes, and not arbitrary objects. Note that in the definition newAB both A and B are *traits*, whereas in the definition of merge the variables x and y denote *objects*.

This limitation essentially put intersection types in Scala in a second-class status. Although merge returns an intersection type, it is hard to actually build values with such types. In essense an object-level introduction contruct for intersection types is missing. As it turns out using low-level type-unsafe programming features such as dynamic proxies, reflection or other meta-programming techniques, it is possible to implement such an introduction construct in Scala [16, 21]. However, this is clearly a hack and it would be better to provide proper language support for such a feature.

2.2 Parametric Polymorphism and Intersection Types

BRUNO: Then talk about parametric polymorphism

Both universal quantification and intersection types provide a kind of polymorphism. While the former provides parametric polymorphism, the latter provides ad-hoc polymorphism. In some systems, parametric polymorphism is considered the infinite analog of intersection polymorphism. But in our system we do not consider this relationship. GEORGE: Need to argue that why their coexistence might be a good thing. GEORGE: May use the merge example

BRUNO: Some more examples in following subsections?

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&B. For example, $1,,\text{`c}'$ is of type Int&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?

The following shows the naive subtyping rules for intersection types:

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

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

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

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;

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

[2] `http://ceylon-lang.org/`

- 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 $\mathsf{Int}\&\mathsf{Int}$ 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:

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

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:

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

x in the body is of type $\alpha\&\mathsf{Int}$ 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:

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

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

Types	A, B	::=	α
		|	$A \to B$
		|	$\forall\alpha.\, A$
		|	$A\&B$
Terms	e	::=	x
		|	$\lambda x{:}A.\, e$
		|	$e_1\ e_2$
		|	$\Lambda\alpha.\, e$
		|	$e\ A$
		|	$e_1,, e_2$
Contexts	Γ	::=	$\cdot \mid \Gamma, \alpha \mid \Gamma, x{:}A$

Figure 1. Syntax.

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. $A\&B$ 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 $A\&B$.

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 [7], and Frisch et. al [13]). 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, $A <: A$.

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

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

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

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

3

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

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 [9]. 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,

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

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

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

It will be translated to

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

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,

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

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.

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 GEORGE: fig:elab-subtyping shows subtyping with coercions. The judgment

$$A_1 <: A_2 \hookrightarrow C$$

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,

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

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

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

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

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

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

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

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

Figure 4. Target language syntax.

$\boxed{|A| = T}$

$$|\alpha| = \alpha$$
$$|\top| = ()$$
$$|A_1| \to |A_2| = |A_1| \to |A_2|$$
$$|\forall\alpha.\, A| = \forall\alpha.\, |A|$$
$$|A_1 \& A_2| = (|A_1|, |A_2|)$$

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

$$|\epsilon| = \epsilon$$
$$|\gamma, \alpha| = |\gamma|, \alpha$$
$$|\gamma, \alpha{:}A| = |\gamma|, \alpha{:}|A|$$

Figure 5. Type and context translation.

generates a coercion function from Int&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* $\epsilon \vdash C : |A_1| \to |A_2|$.

Proof. By a straighforward induction on the derivation[4]. □

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.

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

Although the system shown in the previous section is type-safe, it is not coherent. This section shows how to amend 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. The full specification of the language can be found in the appendix.

Theorem 3 (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.

GEORGE: Transition and informally motivates the following definiton here.

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

Definition 1 (Disjoint types)**.** Two types A and B are *disjoint* if they do not share a common supertype, that is, there does not exist a type C such that $A <: C$ and that $B <: C$. [5]

Given this definiton, now may be the right time to take a short digression to consider union types. If a type system ever contains union types (the counterpart of intersection types), with the following standard subtyping rules,

$$\frac{}{A <: A|B}\ \textsc{Union-1} \qquad \frac{}{B <: A|B}\ \textsc{Union-2}$$

then no two types A and B can ever be disjoint, since there always exists the type $A|B$, which is their common supertype. This serve as the motivation that our system does not permit union types.

To see this definition in action, Int and String are, because there is no type that is a supertype of the both. On the other hand, Int is not disjoint with itself, because $\mathsf{Int} <: \mathsf{Int}$. This implies that disjointness is not reflexive as subtyping is. Two types with different shapes are disjoint unless one of them is an intersection type. For example, a function type and a forall type must be disjoint. But a function type and an intersection type may not be. Consider:

$$\mathsf{Int} \to \mathsf{Int} \quad \text{and} \quad (\mathsf{Int} \to \mathsf{Int})\&(\mathsf{String} \to \mathsf{String})$$

They are not disjoint since $\mathsf{Int} \to \mathsf{Int}$ is their common supertype.

Now that we have obtained a specification for disjointness, but the definition involves an existence problem. How can we implement it? One possibility is bidirectional subtyping, that is, we say two types, A and B, are disjoint if neither $A <: B$ nor $B <: A$. However, this implementation is wrong. For example, $\mathsf{Int}\&\mathsf{String}$ and $\mathsf{String}\&\mathsf{Char}$ are not disjoint by specification since String is their common supertype. Yet by the implementation they are, since neither of them is a subtype of the other. Hence the algorithmic rules are more nuanced and will be developed in the next section. For now, it is enough to treat this judgement as oracle.

GEORGE: Highlight new stuff GEORGE: Discuss each point in this order: goal, idea of tweaks, changes, and proofs.

4.1 Syntax

Figure 4.1 shows the updated syntax with the changes highlighted. There are two changes. First, type variables are always associated with their disjointness constraints, for example $\alpha * A$, in types, terms, and contexts. Second, the bottom type is introduced so that universal quantification quantification be integrated and become a special case of disjoint quantification. That is, $\Lambda\alpha.\, e$ is just a syntactic sugar for $\Lambda\alpha * \bot.\, e$. The underlying idea is that any type is disjoint with the bottom type.

The top type is the trivial upper bound in bounded quantification, while the bottom type is the trivial disjointness constraint in disjoint quantification.

4.2 Typing

GEORGE: Show a diagram here to contrast with bounded polymorphism GEORGE: Missing the merge rule, as it is also changed.

Disjoint quantification is introduced by the big lambda $\Lambda\alpha * A.\, e$ and eliminated by the usual type application $e\ A$. During the type application, the type system makes sure that the type argument satisfies the disjointness constraint. The typing rule for disjoint quantification generalises the rule for quantification in the same way that the rule for bounded quantification does that for quantification.

[5] The definition of disjointness can also be adapted to type systems with a top type (such as Object in many OO languages): Two types A and B are *disjoint* if: the fact that C is a commmon supertype of theirs implies C is the top type.

Types	A, B	$::=$	α
		$\mid$	$\bot$
		$\mid$	$A \to B$
		$\mid$	$\forall\alpha * B.\, A$
		$\mid$	$A\&B$
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 6. Syntax.

$$\frac{\Gamma, \alpha * B \vdash e : A \hookrightarrow E \qquad \Gamma \vdash B\ \text{type} \qquad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash \Lambda\alpha * B.\, e : \forall\alpha * B.\, A \hookrightarrow \Lambda\alpha.\, E}\ \text{T\_BLAM}$$

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

Well-formed Types

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

$$\frac{\Gamma \vdash A\ \text{type} \qquad \Gamma \vdash B\ \text{type} \qquad \Gamma \vdash A * B}{\Gamma \vdash A\&B\ \text{type}}\ \text{WF\_INTER}$$

We additionally require that the two types of an intersection must be disjoint in their context, and that the disjointness constraint in a forall type is well-formed. Under the new rules, intersection types such as $\mathsf{Int}\&\mathsf{Int}$ are no longer well-formed because the two types are not disjoint.

Since in this section we only restrict the system in the previous section, it is easy to see that type preservation and type-safety still holds.

4.3 Subtyping

GEORGE: Two points are being made here: 1) nondisjoint intersections, 2) atomic types. Show an offending example for each?

First we need to introduce the concept of *atomic types*. Atomic types are just those which are not intersection types, and are asserted by the judgement

$$A\ \text{atomic}$$

The complete rules can be found in the appendix.

The subtyping rules, without the atomic condition are overlapping. If we would like to have a deterministic elaboration result, the overarching idea is to tweak the rules so that given a term, it is no longer possible that both of the twin rules can be used. For example, if $A_1\&A_2 <: A_3$, we would like to be certain that either $A_1 <: A_3$ holds or $A_2 <: A_3$ holds, but not both.

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.GEORGE: Point to proofs and justify why the proof shows this.

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

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

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, Int&Int <: Int, obviously the theorem will not hold since both the left Int and the right Int are a subtype of Int.

If our types include primitive subtyping such as $\mathsf{Nat} <:_{\text{prim}} \mathsf{Int}$ (a natural number is also an integer), which can be promoted to the normal subtyping with this rule:

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

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

We can try to rule out such possibilities by making the requirement of well-formedness stronger. This suggests that the two types on the sides of & should not "overlap". In other words, they should be "disjoint". It is easy to determine if two base types are disjoint. For example, Int and Int are not disjoint. Neither do Int and Nat. Also, types built with different constructors are disjoint. For example, Int and Int → 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 Int <: Int&Char to hold in order to get the premise, which does not. So it can be shown that $(\mathsf{Int\&Char})((1,,'c') : \mathsf{Int\&Char}) \hookrightarrow 1$ is not derivable.

4.4 Metatheory

4.4.1 Coherence

Lemma 4 (Unique subtype contributor)**.** *If* A&B <: C*, where* A&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&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.

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

Lemma 5 (Unique coercion)**.** *If* $A <: B \hookrightarrow C_1$ *and* $A <: B \hookrightarrow C_2$*, where* A *and* B *are well-formed types, then* $C_1 \equiv C_2$.

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 6. *Instantiation*

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

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

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

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

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

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

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

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

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

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

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

Figure 7. Algorithmic Disjointness.

Lemma 7. *Well-formed typing If* $\Gamma \vdash e : A$*, then* $\Gamma \vdash e$ *type.*

Typing always produces a well-formed type.

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 (&). (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 8. *Symmetry of disjointness If* $\Gamma \vdash A * B$, *then* $\Gamma \vdash B * A$.

Proof. Trivial by the definition of disjointness. □

Theorem 4. *If* $\Gamma \vdash A * C$ *and* $\Gamma \vdash B * C$, *then* $\Gamma \vdash A\&B * C$.

Lemma 9. *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 5. *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 9
GEORGE: May need an extracted lemma here
- Case
$$\frac{\Gamma \vdash A *_I C \qquad \Gamma \vdash B *_I C}{\Gamma \vdash A\&B *_I C}\ \textsc{DisjointInter1}$$
By Lemma 4 and the i.h.
- Case
$$\frac{\Gamma \vdash A *_I B \qquad \Gamma \vdash A *_I C}{\Gamma \vdash A *_I B\&C}\ \textsc{DisjointInter2}$$
By Lemma 4, Lemma 8, 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 $C = C_1\&C_2$ By inversion and the i.h. we arrive at a contradiction.

□

Theorem 6. *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. GEORGE: ???
  - Case $B = B_1 \to B_2$ The conclusion is true by the disjoint axioms.
  - Case $B = B_1\&B_2$. Need to show $\Gamma \vdash \bot * B_1\&B_2$ implies $\Gamma \vdash \bot *_I B_1\&B_2$. 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 $\Gamma \vdash A_1 \to A_2 *_I B_1 \to B_2$. Apply (DisjointFun) and the result, $\Gamma \vdash A_2 *_I B_2$, can be proved by the i.h.
  - Case $B = B_1\&B_2$. Need to show $\Gamma \vdash A_1 \to A_2 * B_1\&B_2$ implies $\Gamma \vdash A_1 \to A_2 *_I B_1\&B_2$. Apply (DisjointInter2) and the resulting conditions can be proved by the i.h.
- $A = A_1\&A_2$ By (DisjointInter1) and by the i.h.

□

6. Discussion

6.1 Normalising Types

Since the order of the two types in a binary intersection does not matter, we may normalise them to avoid unnecessary coercions.

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 [18] 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 [19] and a 1997 report [20]. Going in the direction of higher kinds, Compagnoni and Pierce [4] 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 [7] 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. [5]. As emphasized throughout the paper our work is inspired by Dunfield [9]. 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 [22] 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 [6, 8, 12] 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. [13] 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* [10], 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* [11] 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, 14, 15, 23]. The work by Nystrom on *nested intersection* [15] uses a form of intersection types to support the composition of families of classes. Ostermann's *delegation layers* [17] 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 Future 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] A. B. Compagnoni and B. C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 1996.

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

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

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

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

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

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

[21] 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.

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

[23] 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.

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 8. Target type system.