Disjoint Intersection Types

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 $F_{\&}$: an extension of System F with *intersection types* and a *merge operator*. The goal of System $F_{\&}$ 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 $F_{\&}$ uses the merge operator to do dynamic composition of values/objects. Moreover, in System $F_{\&}$ type inheritance is independent of subtyping, and an extension can be a supertype of a base object type. Finally, System $F_{\&}$ replaces many uses of generics by intersection types or conventional subtyping. System $F_{\&}$ is formalized and implemented. Moreover the paper shows how various extensible designs can be encoded in System $F_{\&}$.

1. Introduction

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

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

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

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

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

1.1 "Testsuite" of examples

1. $\lambda(x : \mathtt{Int} * \mathtt{Int}).(\lambda(z : \mathtt{Int}).z)\ x$: This example should not type-check because it leads to an ambigous choice in the body of the lambda. In the current system the well-formedness checks forbid such example.
2. $\Lambda A.\Lambda B.\lambda(x : A).\lambda(y : B).(\lambda(z : A).z)(x,,y)$: This example should not type-check because it is not guaranteed that the instantiation of A and B produces a well-formed type. The TyMerge rule forbids it with the disjointness check.
3. $\Lambda A.\Lambda B * A.\lambda(x : A).\lambda(y : B).(\lambda(z : A).z)(x,,y)$: This example should type-check because B is guaranteed to be disjoint with A. Therefore instantiation should produce a well-formed type.
4. $(\lambda(z : \mathtt{Int}).z)((1,,'c'),,(2,\mathtt{False}))$: This example should not type-check, since it leads to an ambigous lookup of integers (can either be 1 or 2). The definition of disjointness is crutial to prevent this example from type-checking. When type-checking the large merge, the disjointness predicate will detect that more than one integer exists in the merge.
5. $(\lambda(f : \mathtt{Int} \rightarrow \mathtt{Int}\&\mathtt{Bool}).\lambda(g : \mathtt{Int} \rightarrow \mathtt{Char}\&\mathtt{Bool}).((f,,g) : \mathtt{Int} \rightarrow \mathtt{Bool})$: This example should not type-check, since it leads to an ambigous lookup of functions. It shows that in order to check disjointness of functions we must also check disjointness of the subcomponents.
6. $(\lambda(f : \mathtt{Int} \rightarrow \mathtt{Int}).\lambda(g : \mathtt{Bool} \rightarrow \mathtt{Int}).((f,,g) : \mathtt{Bool}\&\mathtt{Int} \rightarrow \mathtt{Int})$: This example shows that whenever the return types overlap, so does the function type: we can always find a common subtype for the argument types.

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

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

Figure 1. Atomic types.

1.2 Achieving coherence

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

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

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

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

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

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

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

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

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

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

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

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

2. Introduction

The benefit of a merge, compared to a pair, is that you don't need to explicitly extract an item out. For example, fst (1,'c')

Definition 1. Determinism If $e : \tau_1 \hookrightarrow E_1$ and $e : \tau_2 \hookrightarrow E_2$, then $\tau_1 = \tau_2$ and $E_1 = E_2$.

Coherence is a property about the relation between syntax and semantics. We say a semantics is *coherent* if the syntax of a term uniquely determines its semantics.

Definition 2. Coherence If $e_1 : \tau_1 \hookrightarrow E_1$ and $e_2 : \tau_2 \hookrightarrow E_2$, $E_1 \Downarrow v_1$ and $E_2 \Downarrow v_2$, then $v_1 = v_2$.

Definition 3. Disjointness Two types A and B are *disjoint* (written as "$*$AB") if there does not exist a type C such that C <: A and C <: B and C <: A ∩ B.

2.1 Equational reasoning

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

```
let fst A B (x : A & B) = (\(y : A). y) x in ...
```

Then we have the following equational reasoning:

```
fst Int Int (2,,3)
(\(y : Int). y) (2,,3)
```

2.2 Discussion

In our type-directed translation, some inference rules return conclusions having *the same constructor*. This phenomenon makes the translation nondeterministic. As an example,

```
({x=1},,{x=2}).x
```

can evaluate to either 1 or 2 (according their translation in the target language). In this case, the constructor is the intersection operator, for which both rules, (select1) and (select2), are applicable.

One remedy, which you may have realised, is to enforce the order of applying rules. Whenever the case as shown above happens, the right component of & and ,, will take precedence. In other words, the (select2) rule is tried first. Only if (select2) fails, the (select1) rule is tried. Therefore, ({x=1},,{x=2}).x can only evaluate to 2. Likewise, ({x=1},,{x="hi"}).x will evaluate to "hi" and will be of type String. Generally, three pairs of rules in our system that cause nondeterminism can all be implemented in the same fashion (sub-and2 is favored over sub-and1), and (restrict2 is favored over restrict1).

This approach seem works fine until you think about how it interact with parametric polymorphism.

```
(/\A. \(x:A&Int). x) Int (1,,2) + 1
```

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 $\tau_1 \cap \tau_2 <: \tau_3$, we would like to be certain that either $\tau_1 <: \tau_3$ holds or $\tau_2 <: \tau_3$ holds, but not both.

Formally, we can state this theorem as:

Theorem 1. *If* τ_1, τ_2, τ_3, *and* $\tau_1 \cap \tau_2$ *are well-formed types, and* $\tau_1 \cap \tau_2 <: \tau_3$, *then* $\tau_1 \cap \tau_3$ exclusive *or* $\tau_2 \cap \tau_3$.

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 τ_1 and τ_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 Nat $<:_{\text{prim}}$ Int (a natural number is also an integer), which can be promoted to the normal subtyping with this rule:

$$\frac{\tau_1 <:_{\text{prim}} \tau_2}{\tau_1 <: \tau_2}$$

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.

We shall give two semantics and show the two are the same.

- an type-directed semantics
- a direct operational semantics

say the example above:
without the cast, you could either get: 1,,'c' or 1 depending on what rules you use
but I think with your change, you can only get the first
(which is what we want)
let me see how we can get '1' before the change

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 (Int ∩ Char)((1,,'c') : Int ∩ Char) ↪ 1 is not derivable.

3. 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* [43]: 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 [22, 35, 40]; new programming languages or language extensions [3, 25, 27, 39], and *design patterns* that can be used with existing mainstream languages [12, 28, 41, 44].

Some of the more recent work on extensibility is focused on design patterns. Examples include *Object Algebras* [28], *Modular Visitors* [12, 41] or Torgersen's [41] 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 [35] or Attribute Grammars [24] using Object Algebras [29, 36]. 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 [12, 41]. 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. [10] 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 [12, 29, 36, 41, 44]. Such designs rely on advanced features of generics, such as *F-bounded polymorphism* [5], *variance annotations* [23], *wildcards* [42] and/or *higher-kinded types* [26] 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 [37] with intersection types and a merge operator [16]. 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 [16]. 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 language, allowing encodings for many other language constructs as syntactic sugar. System $\mathsf{F}_{\&}$ 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 $\mathsf{F}_{\&}$ 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 $\mathsf{F}_{<:}$ [7]), System $\mathsf{F}_{\&}$ 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 $\mathsf{F}_{\&}$, capable of expressing various extensibility designs in the literature. System $\mathsf{F}_{\&}$ also addresses limitations of existing OOP languages that complicate extensible designs.
- **Formalization of System $\mathsf{F}_{\&}$:** An elaboration semantics of System $\mathsf{F}_{\&}$ into System F is given, and type-soundness is proved.
- **Encodings of Extensible Designs:** Various encodings of extensible designs into System $\mathsf{F}_{\&}$, 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 $\mathsf{F}_{\&}$, as well as the examples presented in the paper, are publicly available[1].

4. An Overview of System $\mathsf{F}_{\&}$

This section provides the reader with the necessary intuition for $\mathsf{F}_{\&}$ by informally introducing the main features of the language. The features of $\mathsf{F}_{\&}$ are also contrasted with features of mainstream languages such as Java or Scala. Note that this section uses some common syntactic sugar in programming languages, which is part of our implementation.

4.1 Intersection Types in Existing Languages

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
```

[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/`

is rejected by the Scala compiler. The problem is that the with construct for Scala terms 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 [29, 36]. However, this is clearly a hack and it would be better to provide proper language support for such a feature.

4.2 Intersection Types in $F_{\&}$

To address the limitations of intersection types in languages like Scala, $F_{\&}$ allows intersecting any two terms at run time using a *merge* operator (denoted by ,,) [16]. With the merge operator it is trivial to implement the merge function in $F_{\&}$:

```
let merge[A,B] (x : A) (y : B) : A & B = x ,, y;
```

In contrast to Scala's term-level with construct, the operator ,, allows two arbitrary values x and y to be merged. The resulting type is an intersection of the types of x and y (A & B in this case).

Intersection types for overloading. A typical use-case for intersection types is to do *overloading*. The benefit is that programmers can use the same operation on different types and delegate the task of choosing a concrete implementation to the type system. For example, we can define a show function that takes either an integer or a boolean and returns its string representation. In other words, show is also *both* a function from integers to strings as well as a function from boolean to strings. Therefore, in $F_{\&}$ show should be of following type:

```
show : (Int -> String) & (Bool -> String)
```

Assuming that the following two functions are available:

```
showInt : Int -> String
showBool : Bool -> String
```

The overloaded show function is defined by merging the showInt and showBool using the merge operator:

```
let show = showInt ,, showBool;
```

To illustrate the usage, consider the function application show 100. The type system will pick the first component of show, namely showInt, as the implementation being applied to 100 because the type of showInt is compatible with 100, but showBool is not. This example shows that one may regard intersections in our system as "implicit pairs" whose introduction is explicit by the merge operator and elimination is implicit (with no source-level construct for elimination).

Subtyping. The previous example exploits a natural subtyping relation on intersection types. That is the type

```
(Int -> String) & (Bool -> String)
```

is a *subtype* of both Int -> String and Bool -> String. This is why show can take 100 as its argument. Generally speaking an intersection type A & B is a subtype of both A and B. Moreover, subtyping of intersection types in $F_{\&}$ is purely structural and it enjoys of properties such as *idempotence*, *commutativity* and *associativity* (equality is defined as bidirectional subtyping relation):

Idempotent	$A \cap A$	$=$	A
Commutative	$A \cap B$	$=$	$B \cap A$
Associative	$(A \cap B) \cap C$	$=$	$A \cap (B \cap C)$

Type-inheritance In $F_{\&}$ type-inheritance is decoupled from subtyping. Type-inheritance, as formally defined by Cook et al. [10], is:

$$G = F + \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

Here G is a new type defined by adding fields to an existing type F. The argument that "Inheritance is not subtyping" means that structually, G can be: 1) a subtype of F, 2) a supertype of F, or 3) G and F can have no relation at all! If none of $l_1, \ldots, l_n$ is present in F, G can be safely regarded as a subtype of F, since G always contains the fields required by the signature F; if all of $l_1, \ldots, l_n$ are present in F and each of their types is a supertype of the corresponding type, G becomes instead a supertype of F; Conceivably, the third possibility will make F and G have no relation. However, most OO languages conflate the subtyping and type-inheritance, allowing only 1). This leads to cases where the type system prevents expressing semantically correct programs. For example, consider a program that consists of the two types:

```
type A = {x1: Int};
type F = {f: A -> Int};
```

Later one wishes to add a field to form a new type B and refine the parameter type of f by using B instead:

```
type B = {x1: Int, x2: Int}; -- B <: A
type G = {f: B -> Int};      -- G :> F
```

Clearly, B should be a subtype of A. But it is also desirable to have the property that G is a *supertype* of F, since the refinement happens at a parameter position. In OO languages such as Java and Scala, the idiom of making G type-inherit F is through G extends F, which only works when G is a subtype of F.

Although in nominal systems there is the well-known tension between inheritance and subtyping, it is worth noting that extending this benefit to nominal subtyping is possible. Ostermann [31] proposed a flexible nominal system that untangles inheritance and subtyping. For example, users may declare supertype of an existing class type while at the same time inheriting the implementation, and vice versa.

4.3 Generalized Records with Intersection Types

Following Reynolds [38] and Castagna et al. [8], $F_{\&}$ leverages intersection types to type extensible records. The idea is that a multi-field record can be encoded as merges of single-field records, and multi-field record types as intersections. Therefore in $F_{\&}$, there are only single-field record constructs.

Conventionally, record operations work only on record types. But in $F_{\&}$ such operations can occur on *any* type. The reason is that multi-field records in $F_{\&}$ do not have a proper record types. Instead, their types are intersections.

4.3.1 Record Operations

To illustrate the various operations on records, we consider a record with three fields and with the following type:

```
{open : Int, high : Int, low : Int}
```

Note that this type is just syntactic sugar for:

```
{open : Int} & {high : Int} & {low : Int}
```

That is a multi-field record type is desugared as intersections of single-field record types.

$F_{\&}$ has three primitive operations related to records: *construction*, *selection*, and *restriction*. *Extension*, described in many other record systems, is delegated to the merge operator. Working with records is type-safe: the type system prevents accessing or removing a field that does not exist.

Construction. The usual notation for constructing records

```
{open=192, high=195, low=189}
```

is a shorthand for merges of single-field records

```
{open=192} ,, {high=195} ,, {low=189}
```

Selection. Fields are extracted using the dot notation. For example,

```
{open=192, high=195, low=189}.open
```

selects the value of the field labelled open from the record.

Restriction. Restriction $e \setminus l$ removes a field l from an term e. If e contains multiple fields labelled l, only the last field will be removed. Restriction was chosen to be a primitive because other common record operations can be defined in terms of restriction. For example, we can encode record *update* as a restriction followed by a merge. The following example shows updating the high field of a quote:

```
{open=192, high=195, low=189} \ high ,, {high=196}
```

Similarly, *renaming* of a field can be simulated by:

```
{open=192, high=195, low=189} \ high ,, {dayHigh=196}
```

Extension. Extension, just as construction, is performed with the merge operator (,,). The following, for example, adds a close field to the record:

```
{open=192, high=195, low=189} ,, {close=195}
```

Generalized records. In addition, a record is just a normal term and can be merged with any other term, for example, the following program is valid:

```
let mixed: Int & {x: Int} = 1 ,, {x=2};
```

and it is still possible to use record operations on mixed. That is because a record type of the form $\{l : \tau\}$ can be thought as a normal type τ tagged by the label l. For instance, mixed.x extracts the value of the field x from mixed.

4.3.2 Record Subtyping

Subtyping of record types is both in width and in depth, as one might expect: {x: Int, y: Int} is a subtype of {x: Int}; and if T1 is a subtype of T2, then {l: T1} is also a subtype of {l: T2}.

4.4 Intersection Types and Parametric Polymorphism

The combination of intersection types and parametric polymorphism makes System $F_{\&}$ quite expressive. In particular this combination enables an encoding of a simple form of *bounded universal quantification* [6].

Bounded quantification and loss of information. The idea of bounded universal quantification was discussed in the seminal paper by Cardelli and Wegner [6]. They show that bounded quantifiers are useful because they are able to solve the "loss of information" problem. The extension of System F with intersection types is able to address the same problem effectively. Suppose we have the following definitions:

```
let user = {name = "George", admin = true};
let id(user: {name: String}) = user;
```

Under a structural type system, passing user to id is allowed. In other words, the type of user is of a subtype of the expected parameter type of id. However there is a problem: what if programmers want to access the `admin` field later. For example:

```
(id user).admin
```

They cannot do so as the above will not typecheck. After going through the function, the resulting value has the type:

```
{name: String}
```

This is rather undesired because the value does have an `admin` field!

Bounded polymorphism enables the id function to return the exact type of the argument so that there is no problem in accessing the `admin` field later. Consider the example below:

```
let id[A <: {name: String}] (user: A) = user;
(id [{name: String, admin: Bool}] user).admin
```

This piece of pseudo-code, which is not valid in $F_{\&}$ due to the use of bounded polymorphism, illustrates the idea. Instead of giving the user argument a concrete type, the id function specifies that such an argument is a subtype of {name: String}. With such a type the function id avoids losing information about the argument type.

Encoding bounded polymorphism in $F_{\&}$**.** $F_{\&}$ does not have bounded polymorphism. However the same effect can be achieved with a combination of intersection types and parametric polymorphism:

```
let id[A] (user: A & {name: String}) = user;
(id [{admin: Bool}] user).admin
```

By requiring the type of the argument to be an intersection type of a type parameter and the upper bound and passing the type information, we make sure that we can still access the admin field later.

Therefore, this technique allows $F_{\&}$ to encode a simple form of bounded quantification. This is good because it means that $F_{\&}$ can express many common idioms that require bounded quantification without complicating the core calculus with native support for bounded quantifiers.

5. Applications to Extensibility

This section shows that, although $F_{\&}$ is a minimal language, its features are enough for encoding extensible designs that been presented in mainstream languages. Moreover $F_{\&}$ addresses limitations of those languages, making those designs significantly simpler. There are two main advantages of $F_{\&}$ over existing languages:

1. $F_{\&}$ supports dynamic composition of intersecting values.
2. $F_{\&}$ does not couple type inheritance and subtyping. Moreover $F_{\&}$ supports contravariant parameter types in the subtyping relation.

These two features avoid the use of low-level programming techniques, and make the designs less reliant on advanced features of generics.

5.1 Object Algebras

Oliveira and Cook [28] proposed a design pattern that can solve the Expression Problem in languages like Java. An advantage of the pattern over previous solutions is that it is relatively lightweight in terms of type system features. In a latter paper, Oliveira et al. [29] noted some limitations of the original design pattern and proposed some new techniques that generalized the original pattern, allowing it to express programs in a Feature-Oriented Programming [35] style. Key to these techniques was the ability to dynamically compose object algebras.

Unfortunatelly, dynamic composition of object algebras is non-trivial. At the type-level it is possible to express the resulting type of the composition using intersection types. Thus, it is still possible to solve that part problem nicely in a language like Scala (which has

basic support for intersection types). However, the dynamic composition itself cannot be easily encoded in Scala. The fundamental issue is that Scala lacks a merge operator (see the discussion in Section 4.1). Although both Oliveira et al. [29] and Rendell et al. [36] have shown that such a merge operator can be encoded in Scala, the encoding fundamentally relies in low-level programming techniques such as dynamic proxies, reflection or meta-programming.

Because $F_{\&}$ supports a merge operator natively, dynamic object algebra composition becomes easy to encode. The remainder of this section shows how object algebras and object algebra composition can be encoded in $F_{\&}$. We will illustrate this point step-by-step by solving the Expression Problem.

A simple system of arithmetic expressions. In the Expression Problem, the idea is to start with a very simple system modeling arithmetic expressions and evaluation. The initial system considers expressions with two variants (literals and addition) and one operation (evaluation). Here is an interface that supports evaluation:

```
type IEval = {eval : Int};
```

In $F_{\&}$ the interfaces of objects (or object types) are expressed as a record type. A **type** declaration allow us to create a simple alias for a type. In this case IEval is an alias for {eval: Int}.

With object algebras, the idea is to create an object algebra interface, ExpAlg, for expression types with the two variants. This interface has a fixed number of variants, but abstracts over the type of the interpretation E.

```
type ExpAlg[E] = {
  lit : Int -> E,
  add: E -> E -> E
};
```

Having defined the interfaces, we can implement that object algebra interface with evalAlg, which is an object algebra for evaluation.

```
let evalAlg: ExpAlg[IEval] = {
  lit = \(x: Int) -> {eval = x},
  add = \(x: IEval) (y: IEval) -> {
    eval = x.eval + y.eval
  }
};
```

In this example we implement a record, where the two operations lit and add return a record with type IEval. The type ExpAlg[IEval] is the type of object algebras supporting evaluation. However, the one interesting point of object algebras is that other operations can be supported as well.

Add a subtraction variant. The point of the Expression Problem is to support the addition of new features to the existing program, without modifying existing code. The first feature is adding a new variant, such as subtraction. We can do so by simply intersecting the original types and merging with the original values:

```
type SubExpAlg[E] = ExpAlg[E] & {
  sub: E -> E -> E
};
let subEvalAlg = evalAlg ,, {
  sub = \(x: IEval) (y: IEval) -> {
    eval = x.eval - y.eval
  }
};
```

Note that here intersection types are used to model *type inheritance* and the merge operator models a basic form of *dynamic implementation inheritance*.

Add a pretty printing operation. A second extension is adding a new operation, such as pretty printing. Similar to evaluation, the interface of the pretty printing feature is modeled as:

```
type IPrint = {print: String};
```

The implementation of pretty printing for expressions that support literals, addition, and subtraction is:

```
let printAlg: SubExpAlg[IPrint] = {
  lit = \(x: Int) -> {print = x.toString()},
  add = \(x: IPrint) (y: IPrint) -> {
    print = x.print ++ " + " ++ y.print
  },
  sub = \(x: IPrint) (y: IPrint) -> {
    print = x.print ++ " - " ++ y.print
  }
};
```

Usage. With the definitions above, values are created using the appropriate algebras. For example, the expression 7 + 2 is encoded as follows:

```
let e1[E] (f: ExpAlg[E]) =
  f.add (f.lit 7) (f.lit 2);
```

The expressions are unusual in the sense that they are functions that take an extra argument f. The extra argument is an object algebra that uses the functions in the record (lit, add and sub) as factory methods for creating values. Moreover, the algebras themselves are abstracted over the allowed operations such as evaluation and pretty printing by requiring the expression functions to take an extra argument E.

Dynamic object algebra composition. To obtain an expression that supports both evaluation and pretty printing, a mechanism to combine the evaluation and printing algebras is needed. $F_{\&}$ allows such composition: the combine function, which takes two object algebras to create a combined algebra. It does so by constructing a new object algebra where each field is a function that delegates the input to the two algebra parameters.

```
let combine[A,B] (f: ExpAlg[A]) (g: ExpAlg[B]) :
  ExpAlg[A&B] = {
    lit = \(x: Int) -> f.lit x ,, g.lit x,
    add = \(x: A & B) (y: A & B) ->
          f.add x y ,, g.add x y
};

let newAlg = combine[IEval, IPrint] subEvalAlg printAlg;
let o = e1[IEval&IPrint] newAlg;
o.print ++ " = " ++ o.eval.toString()
```

Note that o is a single object that supports both evaluation and printing, as the output of the program is

```
> 7 + 2 = 9
```

In contrast to the Scala solutions available in the literature, $F_{\&}$ is able to express object algebra composition very directly by using the merge operator.

5.2 Back to Visitors

Object Algebras are closely related to the visitor pattern [21]. Indeed, object algebra interfaces are just *internal visitors* [12, 28]. What distinguishes object algebras from the traditional visitor pattern is the lack of a composite interface with an accept method, which is both a blessing and a curse. On the one hand the trouble with composite interfaces with an accept method is that they make adding new variants to the visitor pattern very hard. Although extensible versions of the visitor pattern are possible, they usually require complex types using advanced features of generics [28, 41].

On the other hand, the lack of such composite interfaces makes object algebras harder to use than visitors. As illustrated in Section 5.1, constructing expressions with object algebras can only be done using a function parametrized by an object algebra.

The remainder of the section shows that in $F_{\&}$ there is no need to have a dillema between extensibility using simple types and usability: *in $F_{\&}$ it is possible to have extensible visitors with simple types*! The key to achieve this is to have type inheritance decoupled from subtyping, and allowing contravariant parameter type refinement.

5.2.1 The Problem with Extensible Visitors

We illustrate the problem with extensible visitors using Scala. The type for expressions is defined in Scala as:

```
trait Exp {
  def accept[E](v: ExpAlg[E]): E
}
```

The trait Exp has only one method accept, which takes an internal visitor (or object algebra) as an argument. Here the type ExpAlg [E] is the Scala analogous of the corresponding type defined in Section 5.1 in $F_{\&}$. In terms of the visitor pattern, ExpAlg defines the visit methods for all variants.

Adding a new variant. The difficulties arise when a new variant, such as subtraction is added. To do so an extended visitor interface analogous to SubExpAlg is needed. Moreover a corresponding composite interface SubExp is needed as well:

```
trait SubExp extends Exp {
  def accept[E](v: SubExpAlg[E]): E
}
```

The body of Exp and SubExp are almost the same: they both contain an accept method that takes an object algebra and returns a value of the type E. The only difference in SubExp the object algebra v is of type SubExpAlg[E], which is a subtype of ExpAlg[E].

Inheritance is not subtyping. Since v appears in parameter position of accept and function parameters are naturally contravariant, SubExp[E] should be a *supertype* (and not a subtype) of Exp[E]. However, in Scala every extension induces a subtype. In other words type inheritance and subtyping always go along together. To ensure type-soundness Scala (and other common OO languages) forbids any kind of type-refinement on method parameter types. In other words method parameter types are invariant. The consequence of this is that Scala is not capable of expressing that SubExp [E] is an extension and a supertype of Exp. Such kind of extension is an example where "*inheritance is not subtyping*" [10].

5.3 Extensible Visitors in $F_{\&}$

Such limitation does not exist in $F_{\&}$. For example, we can define the similar interfaces for Exp and SubExp:

```
type Exp = {
  accept: forall E. ExpAlg[E] -> E
};
type SubExp = {
  accept: forall E. SubExpAlg[E] -> E
};
```

$F_{\&}$ support contravariant parameter type refinement, which means that SubExp is a supertype of Exp. Using these types we first define two data constructors for simple expressions:

```
let lit (n: Int): Exp = {
  accept = /\E -> \(f: ExpAlg[E]) -> f.lit n
};
let add (e1: Exp) (e2: Exp): Exp = {
  accept = /\E -> \(f: ExpAlg[E]) ->
    f.add (e1.accept[E] f) (e2.accept[E] f)
};
```

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 2. Syntax.

Both lit and add build values of type Exp and use object algebras of type ExpAlg[E]. However, subtraction requires a value of type SubExp to be created:

```
let sub (e1: SubExp) (e2: SubExp): SubExp = {
  accept = /\E -> \(f: SubExpAlg[E]) ->
    f.sub (e1.accept[E] f) (e2.accept[E] f)
};
```

Usage. With visitors constructing expressions is quite simple:

```
let e1 = sub ( lit 7) ( lit 2);
e1.accept[String] printAlg
```

The result is "7 − 2". Note that the programmer is able to pass lit 2, which is of type Exp, to sub, which expects a SubExp. The types are compatible because because Exp is a *subtype* of SubExp. Code reuse is achieved since we can use the constructors from Exp as the constructor for SubExp. In Scala, we would have to define two literal constructors, one for Exp and another for SubExp.

Compared to object algebras, the addition of the composite structure allows values to be created much more intuitively, without any drawback! All the code developed with object algebras works right away with visitors.

Finally note that in terms of typing, this solution does not require any advanced use of generics. This is in sharp constrast with previous proposals for extensible visitors in the literature.

6. 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.

6.1 Syntax

Figure 6.1 shows the syntax of $F_{\&}$ (with the addition to System F highlighted).

Meta-variables τ range over types. Types include System F constructs: type variables α; function types $\tau_1 \to \tau_2$; and type abstraction $\forall \alpha. \tau$. The bottom type $\bot$ is not inhabited by any term. $\tau_1 \cap \tau_2$ denotes the intersection of types τ_1 and τ_2. We omit type constants such as tyint and tystring .

Terms include standard constructs in System F: variables x; abstraction of terms over variables of a given type $\lambda x{:}\tau. e$ (concretely

written \(x:T) -> e); abstraction of terms over types $\Lambda\alpha.\, e$ (concretely written /\A -> e); application of terms to terms $e_1\ e_2$; and application of terms to types $e\ \tau$ (concretely written e[T]). The last four constructs are novel. The canonical term that inhabits the top type is also written as $\top$. $e_1 ,, e_2$ 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 $\tau_1 \cap \tau_2$.

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 τ. We use use $[\tau_1/\alpha]\ \tau_2$ for the capture-avoiding substitution of τ_1 for α inside τ_2 and $\mathsf{ftv}(\cdot)$ for sets of free variables.

6.2 Subtyping

The subtyping rules of $F_{\&}$, shown in Figure 3, are syntax-directed (different from the approach by Davies and Pfenning [14], and Frisch et. al [20]). 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 τ_1 is both the subset of τ_2 and the subset of τ_3, then τ_1 is also the subset of the intersection of τ_2 and τ_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* τ, $\tau <: \tau$.

Lemma 2 (Subtyping is transitive)**.** *If* $\tau_1 <: \tau_2$ *and* $\tau_2 <: \tau_3$*, then* $\tau_1 <: \tau_3$.

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

6.3 Disjointness and well-formedness

The rules for the disjointness judgement are shown in Figure 6.3. 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).

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

6.4 Typing

The syntax-directed typing rules of $F_{\&}$ are shown in Figure 6. They consist of one main typing judgment and two auxiliary judgments. The main typing judgment is of the form: $\Gamma \vdash e : \tau$. It reads: "in the typing context Γ, the term e is of type τ". The rules that are the same as in System F are rules for variables ((VAR)), lambda abstractions ((LAM)), type abstractions ((BLAM)), 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 (τ_3) is a subtype of that of the parameter (τ_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.

7. 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 [16]. 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.

7.1 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.

7.2 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 7 and the typing rules in the appendix.

7.3 Type translation

Figure 8 defines the type translation function $|\cdot|$ from $F_{\&}$ types τ to target language types T. The notation $|\cdot|$ is also overloaded for context translation from $F_{\&}$ contexts γ to target language contexts Γ.

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

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

$$\frac{\tau_1 <: [\alpha_1/\alpha_2]\,\tau_2 \hookrightarrow C}{\forall \alpha_1 * \tau_3.\, \tau_1 <: \forall \alpha_2 * \tau_3.\, \tau_2 \hookrightarrow \lambda f{:}|\forall \alpha_1 * \tau_3.\, \tau_1|.\, \Lambda \alpha.\, C\ (f\ \alpha)}\ \textsc{SubForall} \qquad \frac{\tau_1 <: \tau_2 \hookrightarrow C_1 \qquad \tau_1 <: \tau_3 \hookrightarrow C_2}{\tau_1 <: \tau_2 \cap \tau_3 \hookrightarrow \lambda x{:}|\tau_1|.\, (C_1\ x, C_2\ x)}\ \textsc{SubAnd}$$

$$\frac{\tau_1 <: \tau_3 \hookrightarrow C \qquad \tau_3 \text{ atomic}}{\tau_1 \cap \tau_2 <: \tau_3 \hookrightarrow \lambda x{:}|\tau_1 \cap \tau_2|.\, C\ (\mathsf{proj}_1 x)}\ \textsc{SubAnd}_1 \qquad \frac{\tau_2 <: \tau_3 \hookrightarrow C \qquad \tau_3 \text{ atomic}}{\tau_1 \cap \tau_2 <: \tau_3 \hookrightarrow \lambda x{:}|\tau_1 \cap \tau_2|.\, C\ (\mathsf{proj}_2 x)}\ \textsc{SubAnd}_2$$

3

Figure 3. Subtyping in $F_{\&}$.

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

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

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

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

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

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

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

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

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

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

Figure 4. Disjointness.

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

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

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

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

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

Figure 5. Well-formed types.

7.4 Coercive subtyping

fig:elab-subtyping

Figure shows subtyping with coercions. The judgment

$$\tau_1 <: \tau_2 \hookrightarrow C$$

extends the subtyping judgment in Figure 3 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 $|\tau_1| \to |\tau_2|$ (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 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 $\lambda x{:}|\tau_3|.\, f\ x$, 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* $\tau_1 <: \tau_2 \hookrightarrow C$, *then* $\epsilon \vdash C : |\tau_1| \to |\tau_2|$.

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

7.5 Main translation

Main translation judgment. The main translation judgment $\gamma \vdash e : \tau \hookrightarrow E$ extends the typing judgment with an elaborated term on the right hand side of $\hookrightarrow$. The translation ensures that E has type $|\tau|$. 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 2 (Translation preserves well-typing)**.** *If* $\gamma \vdash e : \tau \hookrightarrow E$, *then* $|\gamma| \vdash E : |\tau|$.

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

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

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

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

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

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

Figure 6. 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 7. Target language syntax.

$$\boxed{|\tau| = T}$$

$$|\alpha| = \alpha$$
$$|\top| = ()$$
$$|\tau_1| \to |\tau_2| = |\tau_1| \to |\tau_2|$$
$$|\forall \alpha.\, \tau| = \forall \alpha.\, |\tau|$$
$$|\tau_1 \cap \tau_2| = (|\tau_1|, |\tau_2|)$$

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

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

Figure 8. Type and context translation.

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

Theorem 3 (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. □

8. 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.

9. Disjoint Intersection Types

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.

9.1 Motivating design choices

BRUNO: Maybe this belongs to Section 2?

We need to motivate the 3 changes:

Well-formed types : We need a new notion of well-formed types.

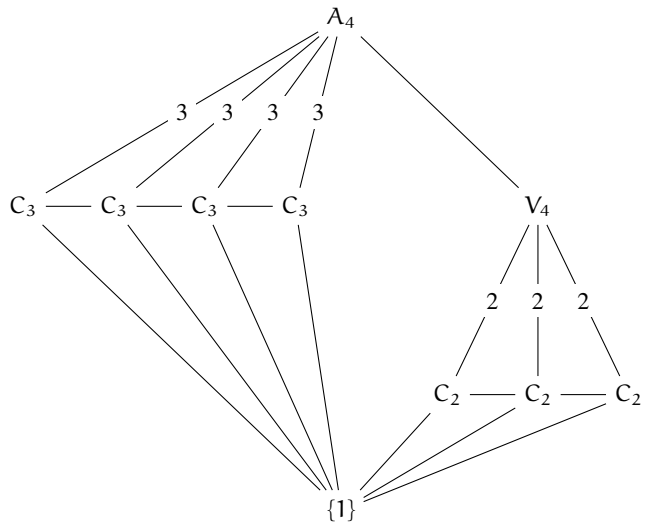

Figure 9. Untergruppenverband

Disjoint quantification : Needed when we have a combination of polymorphism and intersection types.

With a subtyping relation in a type system, bounded polymorphism extends the universal quantifier by confining the polymorphic type to be a subtype of a given type. In our type system, the forall binder also extends the parametric polymorphism, but in a different vein: the polymorphic type can only be disjoint with a given type. Later during an instantiation, if the type provided overlaps with the constraint, such instantiation will be rejected by our type system.

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

The intuition can be found in figure ....

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

GEORGE: Add interpretation of the theorem

Theorem 4. *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), $A \cap B <: C_1 \cap C_2$. □

9.2 Disjointness

Spec of disjointness/intuition ...

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

Definition 4 (Disjointness). $A \perp B = \nexists C. A <: C \wedge B <: C$

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

9.3 Well-formed types

A well-formed type is such that given any query type, it is always clear which subpart the query is referring to. The rules for well-formedness are standard except for intersection types we require the two components to be disjoint.

9.4 Subtyping

9.5 Metatheory

Definition 5. 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 $B = B_1 \cap B_2$[5]: 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:

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

[5] An example of this case is:

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

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
$$\frac{\Gamma \vdash A *_I C}{\Gamma \vdash \forall \alpha * B.\, A *_I \forall \alpha * B.\, \mathrm{CSubst???}}\ \textsc{DisjointForall}$$
By applying the i.h. and the def. of the substitution function. Note that α is fresh.
- 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),

Show disjointness

$$\Gamma \vdash [A/\alpha]\ C_1 \cap [A/\alpha]\ C_2 \text{ type}$$

and hence

$$\Gamma \vdash [A/\alpha]\ (C_1 \cap C_2) \text{ type}$$

□

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 5 (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.

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

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. □

10. Algorithmic Disjointness

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

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

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

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

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

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

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

10.1 Formal System

Explain the rules and intuitions.

11. Discussions

11.1 Systems without subtyping

11.2 Systems with a top type

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

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

12. Metatheory

12.1 Coherence

12.2 Equivalence of disjointness

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

Proof. By induction on $*_I$.

- Case

$$\frac{\Gamma \vdash B *_I D}{\Gamma \vdash A \to B *_I C \to D}\ \text{DISJOINTFUN}$$

  Lemma 12

  GEORGE: May need an extracted lemma here
- Case

$$\frac{\Gamma \vdash A *_I C \qquad \Gamma \vdash B *_I C}{\Gamma \vdash A \cap B *_I C}\ \text{DISJOINTINTER1}$$

  By Lemma 6 and the i.h.
- Case

$$\frac{\Gamma \vdash A *_I B \qquad \Gamma \vdash A *_I C}{\Gamma \vdash A *_I B \cap C}\ \text{DISJOINTINTER2}$$

  By Lemma 6, 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 8. *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 $\Gamma \vdash A_1 \to A_2 *_I B_1 \to B_2$. Apply (DisjointFun) and the result, $\Gamma \vdash A_2 *_I B_2$, can be proved by the i.h.
  - Case $B = B_1 \cap B_2$. 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.

□

13. Related work

Intersection types with polymorphism. Our type system combines intersection types and parametric polymorphism. Closest to us is Pierce's work [32] 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 [33] and a 1997 report [34]. Going in the direction of higher kinds, Compagnoni and Pierce [9] 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 [14] 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 5, 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. [11]. As emphasized throughout the paper our work is inspired by Dunfield [16]. 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 [38] 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 [13, 15, 19] 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. [20] 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* [17], 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* [18] 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, 25, 27, 39]. The work by Nystrom on *nested intersection* [27] uses a form of intersection types to support the composition of families of classes. Ostermann's *delegation layers* [30] 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 $F_{\&}$.

14. 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 and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 1985.

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

[8] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 1995.

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

[10] 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.

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

[12] 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.

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

[14] 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.

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

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

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

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

[19] 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.

[20] 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.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[22] 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.

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

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

[25] 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.

[26] 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.

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

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

[29] 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.

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

[31] K. Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 2008.

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

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

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

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

[36] 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.

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

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.