System $F_{\&}$: A Simple Core Language for Extensibility

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

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* [49]: there are *two* common and desirable forms of extensibility, but most mainstream languages can only support one type well. Unfortunately the lack of support in the other direction has significant consequences in terms of code maintence and software evolution. As a result researchers proposed various approaches to address the problem, including: visions of new programming models [26, 41, 46]; new programming languages or language extensions [3, 31, 33, 45], and *design patterns* that can be used with existing mainstream languages [16, 34, 47, 52].

Some of the more recent work on extensibility is focused on design patterns. Examples include *Object Algebras* [34], *Modular Visitors* [16, 47] or Torgersen's [47] 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, encodings of Feature-Oriented Programming [41] or Attribute Grammars [28] using Object Algebras [35, 42]. 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 meta-programming. It is clear that better language support would be desirable.

The second limitation shows up in designs for modelling modular or extensible visitors [16, 47]. 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. [13] 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 compeling example where inheritance and subtyping should not be conflated!

CONF 'yy, Month d--d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Finally, the third limitation is prevalent in many extensible designs [16, 35, 42, 47, 52]. Such designs rely on advanced features of generics, such as *f-bounded polymorphism* [5], *variance annotations* [27], *wildcards* [48] and/or *higher-kinded types* [32] 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 $F_{\&}$ (pronounced *f-and*): an extension of System F [43] with intersection types and a merge operator [20]. The goal of System $F_{\&}$ 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 $F_{\&}$ uses the merge operator for dynamic composition of values/objects. Moreover, in System $F_{\&}$ (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 how many previous designs in the literature can be encoded without such advanced features of generics.

Technically speaking System $F_{\&}$ is mainly inspired by the work of Dundfield [20]. Dundfield shows 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 $F_{\&}$ 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. In contrast Dunfield has an additional step which eliminates subtyping. Finally a non-technical difference is that System $F_{\&}$ is aimed at studying issues of OOP languages and extensibility, whereas Dunfield's work was aimed at Functional Programming and he did not consider applications to extensibility. Like many other foundational formal models for OOP (for example $F_{<:}$ [9]), System $F_{\&}$ is purely functional and it uses structural typing.

In summary, the contributions of this paper are:

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

2. An Overview of System $F_{\&}$

bruno: Syntax in the examples is not being used consistently! That is why it is better to import code from a file. You have to carefully go over every example and see whether the examples are actually valid syntax in our language. I fixed some of these, but there may be some more.

This section provides the reader with the necessary intuition for $F_{\&}$ by informally introducing the main features of the language. The features of $F_{\&}$ 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 of System $F_{\&}$.

2.1 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
```

is rejected by the Scala compiler. The problem is that the `with` construct for Scala expressions can only be used to

[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/`

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 [35, 42]. However, this is clearly a hack and it would be better to provide proper language support for such a feature.

2.2 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 ,,) [20]. 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 expression-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* and *commutativity* (equality is defined as bidirectional subtyping relation):

Idempotent $A \& A = A$
Commutative $A \& B = B \& A$

george: Assoc? not sure!

Covariance and contravariance. The subtyping also arises from contravariant parameter types and covariant return types for functions. bruno: I think this will be a good place to talk about the following material: bruno: Talk about Inheritance is not Subtyping. Describe type inheritance and subtyping, show that they don't necessarelly go along together in our language. You may need to write some Java code, to illustrate differences. *We support contravariant argument types!* bruno: Related to the previous point, don't forget to mention that there are nominal languages, that also separate inheritance from subtyping! See Klaus Ostermann's paper & ``Inheritance is not Subtyping''.

Our system enjoys modular extensibility that structural subtyping offers. 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 [37] 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. However the price to pay is that the subtyping is not transitive.

george: I think points regarding the designs of type system should prob. be addressed somewhere else. The purpose of this section is just introducing the source language and preparing the reader for the next section.

2.3 Generalized Records with Intersection Types

Following Reynolds [44] and Castagna et al. [10], $F_{\&}$ 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 $F_{\&}$ generalizes them, allowing operations to 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.

2.3.1 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 updating 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 expression e. If e contains multiple fields labelled l, only the last field will be removed. Restriction was chosen to be a primitive because we may define other common record operations in terms of restriction. For example, we can define record *update* as a restriction followed by a merge. The following example creates a new quote with the `high` field updated:

```
{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 the `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`.

2.3.2 Record Subtyping

Subtyping of record types supports both width and 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}` .

2.4 Intersection Types and Parametric Polymorphism

bruno: wondering again whether this should be here or in the next section!

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* [8].

Bounded quantification and loss of information. The idea of bounded universal quantification was discussed in the seminal paper by Cardelli and Wegner [8]. 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 calclus with native support for bounded quantifiers.

3. Applications to Extensibility

bruno: Make sure that the important code in the paper is reused from a script and not inlined directly in the text.

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.

3.1 Object Algebras

Oliveira and Cook [34] 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. [35] 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 [41] 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 2.1). Although both Oliveira et al. [35] and Rendell et al. [42] 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 algebras composition can be encoded in $F_{\&}$. We will illustrate this point with an step-by-step of 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 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 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 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: SubExpAlg[E]) =
   f.sub (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.

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.

3.2 Back to Visitors

Object Algebras are closely related to the visitor pattern [25]. Indeed, object algebra interfaces are just *internal visitors* [16, 34]. 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 [34, 47]. On the other hand, the lack of such composite interfaces makes object algebras harder to use than visitors. As illustrated in Section 3.1, 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.

3.2.1 The Problem with Extensible Visitors

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

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

The trait `Exp` has only one method `accept`, which takes an internal visitor (or object algebra) as an argument. Here the type `ExpAlg[A]` is the Scala analogous of the corresponding type defined in Section **??** 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 {
  override 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 sub-

typing always go along together. To ensure type-soundness Scala (and other common OO languages) forbid 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*`` [13].

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

$F_{\&}$ 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)
};
```

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:

```
e2 = sub (lit 7) (lit 2)
```

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.

4. The $F_{\&}$ calculus

george: Discuss duplicate labels? george: Fix highlighting or just remove it

Following Dunfield's [20] work on a simply-typed lambda calculus with intersection and union types, we present the syntax, subtyping, and typing of $F_{\&}$. The semantics of $F_{\&}$ will be defined by a type-directed translation from $F_{\&}$ to a simple variant of System F in the next section.

4.1 Syntax

Figure 1 shows the syntax of $F_{\&}$ (with the addition to System F highlighted). As a convention in this paper, we will be using lowercase letters as meta-variables for sorts in $F_{\&}$, and uppercase letters for those in the target language (starting to appear in the next section).

Types	τ	$::=$	$\alpha \mid \top \mid \tau_1 \to \tau_2 \mid \forall\alpha.\,\tau$
		$\mid$	$\tau_1 \,\&\, \tau_2 \mid \{l{:}\tau\}$
Expressions	e	$::=$	$x \mid \top \mid \lambda(x{:}\tau).\,e \mid e\ e \mid \Lambda\alpha.\,e \mid e\ \tau$
		$\mid$	$e,,e \mid \{l = e\} \mid e.l \mid e \setminus l$
Contexts	γ	$::=$	$\epsilon \mid \gamma, \alpha \mid \gamma, x{:}\tau$
Labels	l		

Figure 1. Syntax of $F_{\&}$.

bruno: I am not sure if the highlighting will be visible. Use gray?bruno: there is no `with` anymore. Make sure the syntax is consistent with what is presented in the type rules!

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 top type $\top$ is the supertype of all types. It is in fact the 0-ary intersection. $\tau_1 \,\&\, \tau_2$ denotes the intersection of type τ_1 and τ_2; and $\{l : \tau\}$ the types for single-field records. Single-field record types can be viewed as types tagged with a label l, while the other type forms are untagged types. We omit type constants such as `Int` and `String`.

Expressions include standard constructs in System F: variables x; abstraction of expressions over variables of a given type $\lambda(x:\tau).\,e$ (concretely written `\(x:T)-> e`); abstraction of expressions over types $\Lambda\alpha.\,e$ (concretely written `/\A -> e`); application of expressions to expressions $e_1\ e_2$; and application of expressions to types $e\ \tau$ (concretely written `e [T]`). The last four constructs are new. The canonical term that inhabits the top type is also written as $\top$. $e_1,,e_2$ is the *merge* of two expressions 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 expressions correspond to intersection types $\tau_1 \,\&\, \tau_2$. The expression $\{l = e\}$ constructs a single-field record, whereas $e.l$ accesses the field labelled l in e. Restriction $e \setminus l$ removes the field l inside e. 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 and variables (and their type τ). We use $\mathtt{l}$ for labels, whose nature is left undefined. We use $[\tau_1/\alpha]\tau_2$ for the capture-avoiding substitution of τ_1 for α inside τ_2 and $ftv(\cdot)$ for sets of free variables.

4.2 Subtyping

The syntax-directed subtyping rules of $F_{\&}$ are shown in Figure 2. They are not very surprising. The rule **sub**fun says that functions are contravariant in their parameter type and covariant in their return type. A universal quantifier ($\forall$) is covariant in its body. A single-field record type is also covariant, which becomes obvious if we regard it as just a labelled type. 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 **sub**and says that if τ_1 is both the subset of τ_2 and τ_3, then τ_1 is also the subset of the intersection of τ_2 and τ_3.

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. bruno: State the reflexivity and transitivity Lemmas here then!

4.3 Typing

bruno: Please make the rule forms appear slightly above the rules.

The syntax-directed typing rules of $F_{\&}$ are shown in Figure 3. They consists of one main typing judgment and two auxiliary judgments. The main typing judgment is of the form: $\gamma \vdash e : \tau$. It says that ``in the typing context γ, the expression e is of type τ". The rules that are the same as in System F are rules for variables (Evar), lambda abstractions (Elam), type abstraction (Eblam), and type application (Etapp). The rule Eapp needs special attention as we add a subtyping requirement in the premise: 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 respectively, and give it the intersection of the resulting types τ_1 & τ_2. The rule for single-field record construction (Erec-construct) is standard. The rules for record selection (Erec-select) and restriction (Erec-restrict) are expectedly the most complicated. They turn to the auxiliary ``select" and ``restrict" rules to statically check operations and to obtain resulting types.

Typing record selection. The ``select" judgment deals with record selection. For example,

$$\{\mathtt{id{:}Int}\} \,\&\, \{\mathtt{name{:}String}\} \bullet \mathtt{name} = \mathtt{String}$$

That means the `name` field inside $\{\mathtt{id{:}Int}\}\&\{\mathtt{name{:}String}\}$ is of type `String`. Formally, $\tau_1 \bullet \mathtt{l} = \tau_2$ says that a field labelled $\mathtt{l}$ is present inside τ_1 and is of type τ_2. The judgment is made of three inference rules and is recursively defined. **select** is the base case: if we ask for a field labelled $\mathtt{l}$ inside $\{\mathtt{l} : \tau\}$, the field is clearly present and is of type τ. **select**$_1$ and **select**$_2$ are two symmetric step cases. Take **select**$_1$ for example, it means that if $\mathtt{l}$ is present inside τ_1, then it is also present inside τ_1 & τ_2; and the type of the desired field $\mathtt{l}$ remains τ in the conclusion.

Typing record restriction. The ``restrict" judgment $\tau_1 \setminus \mathtt{l} = \tau_2$ deals with record restriction and is very similar to the ``select" judgment. The only difference is that instead of giving the type of field being selected, the judment holds the type after the restriction. For example,

$$\{\mathtt{id{:}Int}\} \,\&\, \{\mathtt{name{:}String}\} \setminus \mathtt{name} = \{\mathtt{id{:}Int}\} \,\&\, \top$$

Note that $\{\mathtt{id} : \mathtt{Int}\}$ & $\top$, interpreted set-theoretically, is equivalent to $\{\mathtt{id{:}Int}\}$. Indeed, our system is able to judge they are subtype of each other.

bruno: We should probably say that id:Int & T is equivalent to id:Int

5. 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 [20]. But in addition, our translation removes labels of records and rewrites record operations as function applications. In the end the translated expressions can be typed and interpreted within System F.

5.1 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,

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

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

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

It will be translated to

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

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

$\boxed{\tau <: \tau}$

$$\frac{}{\alpha <: \alpha}\ \textsc{subvar} \qquad \frac{}{\tau <: \top}\ \textsc{subtop} \qquad \frac{\tau_3 <: \tau_1 \qquad \tau_2 <: \tau_4}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4}\ \textsc{subfun} \qquad \frac{\tau_1 <: [\alpha_1/\alpha_2]\tau_2}{\forall \alpha_1.\, \tau_1 <: \forall \alpha_2.\, \tau_2}\ \textsc{subforall}$$

$$\frac{\tau_1 <: \tau_2 \qquad \tau_1 <: \tau_3}{\tau_1 <: \tau_2 \,\&\, \tau_3}\ \textsc{suband} \qquad \frac{\tau_1 <: \tau_3}{\tau_1 \,\&\, \tau_2 <: \tau_3}\ \textsc{suband}_1 \qquad \frac{\tau_2 <: \tau_3}{\tau_1 \,\&\, \tau_2 <: \tau_3}\ \textsc{suband}_2 \qquad \frac{\tau_1 <: \tau_2}{\{l{:}\tau_1\} <: \{l{:}\tau_2\}}\ \textsc{subrec}$$

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

$\boxed{\gamma \vdash e : \tau}$

$$\frac{(x, \tau) \in \gamma}{\gamma \vdash x : \tau}\ \textsc{Evar} \qquad \frac{}{\gamma \vdash \top : \top}\ \textsc{Etop} \qquad \frac{\gamma, x{:}\tau \vdash e : \tau_1 \qquad \gamma \vdash \tau}{\gamma \vdash \lambda(x{:}\tau).\, e : \tau \to \tau_1}\ \textsc{Elam}$$

$$\frac{\gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \gamma \vdash e_2 : \tau_3 \qquad \tau_3 <: \tau_1}{\gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{Eapp} \qquad \frac{\gamma, \alpha \vdash e : \tau}{\gamma \vdash \Lambda\alpha.\, e : \forall \alpha.\, \tau}\ \textsc{Eblam} \qquad \frac{\gamma \vdash e : \forall \alpha.\, \tau_1 \qquad \gamma \vdash \tau}{\gamma \vdash e\ \tau : [\tau/\alpha]\tau_1}\ \textsc{Etapp}$$

$$\frac{\gamma \vdash e_1 : \tau_1 \qquad \gamma \vdash e_2 : \tau_2}{\gamma \vdash e_1 ,, e_2 : \tau_1 \,\&\, \tau_2}\ \textsc{Emerge} \qquad \frac{\gamma \vdash e : \tau}{\gamma \vdash \{l = e\} : \{l{:}\tau\}}\ \textsc{Erec-construct} \qquad \frac{\gamma \vdash e : \tau \qquad \tau \bullet l = \tau_1}{\gamma \vdash e.l : \tau_1}\ \textsc{Erec-select}$$

$$\frac{\gamma \vdash e : \tau \qquad \tau \setminus l = \tau_1}{\gamma \vdash e - l : \tau_1}\ \textsc{Erec-restrict}$$

$\boxed{\tau_1 \bullet l = \tau_2}$

$$\frac{}{\{l{:}\tau\} \bullet l = \tau}\ \textsc{select} \qquad \frac{\tau_1 \bullet l = \tau}{\tau_1 \,\&\, \tau_2 \bullet l = \tau}\ \textsc{select}_1 \qquad \frac{\tau_2 \bullet l = \tau}{\tau_1 \,\&\, \tau_2 \bullet l = \tau}\ \textsc{select}_2$$

$\boxed{\tau_1 \setminus l = \tau_2}$

$$\frac{}{\{l{:}\tau\} \setminus l = \top}\ \textsc{restrict} \qquad \frac{\tau_1 \setminus l = \tau}{\tau_1 \,\&\, \tau_2 \setminus l = \tau \,\&\, \tau_2}\ \textsc{restrict}_1 \qquad \frac{\tau_2 \setminus l = \tau}{\tau_1 \,\&\, \tau_2 \setminus l = \tau_1 \,\&\, \tau}\ \textsc{restrict}_2$$

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

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

```
{name = "Barbara"}
```

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

```
{distance = {inKilometers = 8, inMiles = 5}}
```

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

```
{distance = {inKilometers = 8} ,, {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

```
{distance = {inKilometers = 8, inMiles = 5}}.
    distance.inMiles
```

becomes

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

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

5.2 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 4 and the typing rules in the Appendix george: cross ref.

Types	T	$::=$	$\alpha \mid () \mid T \to T \mid \forall \alpha.\, T \mid (T, T)$
Expressions	E, C	$::=$	$x \mid () \mid \lambda(x{:}T).\, E \mid E\ E \mid \Lambda\alpha.\, E$
		$\mid$	$E\ T \mid (E, E) \mid \texttt{proj}_k E$
Contexts	Γ	$::=$	$\epsilon \mid \Gamma, \alpha \mid \Gamma, x{:}T$

Figure 4. Target language syntax.

5.3 Type Translation

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

$$
\begin{aligned}
|\alpha| &= \alpha \\
|\top| &= () \\
|\tau_1| \to |\tau_2| &= |\tau_1| \to |\tau_2| \\
|\forall \alpha.\, \tau| &= \forall \alpha.\, |\tau| \\
|\tau_1 \,\&\, \tau_2| &= (|\tau_1|, |\tau_2|) \\
|\{l{:}\tau\}| &= |\tau|
\end{aligned}
$$

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

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

Figure 5. Type and context translation.

Figure 5 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.

5.4 Coercive Subtyping

Figure 6 shows subtyping with coercions. The judgment

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

extends the subtyping judgment in Figure 2 with a coercion on the right hand side of $\hookrightarrow$. A coercion C is just an expression in the target language and is ensured to have type $|\tau_1| \to |\tau_2|$ (Lemma **??**)bruno: ref now showing. For example,

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

generates a coercion function from `Int & Bool` to `Bool`.

In rules **sub**var, **sub**top, **sub**forall, coercions are just identity functions. In **sub**fun, 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 **sub**and$_1$, **sub**and$_2$, and **sub**and elaborate with intersection types. **sub**and uses both coercions to form a pair. Rules **sub**and$_1$ and **sub**and$_2$ 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 (**sub** 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. □

5.5 Main Translation

Main translation judgment. The main translation judgment $\gamma \vdash e : \tau \hookrightarrow E$ extends the typing judgment with an elaborated expression 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 **E**app, the coercion C from the subtyping relation is applied to the argument. **E**merge straighforwardly translates merges into pairs. As record labels are erased, **E**rec-construct yields the same target expression E from the premise.

Erec-select typechecks e and use the ``select'' rule to return the type of the field $\tau_1$ and the coercion C. The type of the whole expression is $\tau_1$ and its translation of C E. **E**rec-restrict is exactly the same as **E**rec-select except that it uses the auxiliary ``restrict'' rule.

***``Select'' judgment.*** The ``select'' judgment additionally generates a coercion on the right-hand side of $\hookrightarrow$, which can be thought as a field selector in the target language. For example, in translating the $F_{\&}$ expression

$$\{\texttt{id} = 12\}.\texttt{id}$$

the judgment

$$\{\texttt{id:Int}\} \bullet \texttt{id} = \texttt{Int} \hookrightarrow \lambda(x{:}|\{\texttt{id:Int}\}|).\, x$$

gives a ``selector'' $\lambda(x{:}|\{\texttt{id:Int}\}|).\, x$ that can be applied to the translation of $\{\texttt{id} = 12\}$. The generation of selectors is defined recursively. **select** is the base case: since the type of the field labelled l in a $\{l : \tau\}$ is just τ, the coercion is an identity function. **select**$_1$ and **select**$_2$ builds selectors for intersection types $\tau_1 \,\&\, \tau_2$ according to the selector for τ_1 or τ_2. The same idea appears in the twin **sub**and$_1$ and **sub**and$_2$.

Lemma 4 (**select** rules produce type-correct coercion)**.** *If* $\tau \bullet l = \tau_1 \hookrightarrow C$, *then* $\epsilon \vdash C : |\tau| \to |\tau_1|$.

Proof. By structural induction of the derivation. □

***``Restrict'' judgment.*** The ``restrict'' judgment deals with record restriction. The rules are analogous to the ``select'' rules. Compared with the coercions generated by the ``select'' rules, the coercions generated here keep all but the restricted field in an expresison. In the base case (**restrict**), removing a field labelled l from a single-field record with the same label should result in the top value. Therefore, the coercion is a constant function that returns unit, which is just the image of top value in the target language. For the case

$$\{\texttt{name} = \texttt{"Alan"}\} \,\&\, \{\texttt{age} = 24\} \setminus \texttt{name}$$

$$\boxed{\tau <: \tau \hookrightarrow C} \qquad \frac{}{\alpha <: \alpha \hookrightarrow \lambda(x{:}|\alpha|).\, x}\ \textbf{sub}\text{var} \qquad \frac{}{\tau <: \top \hookrightarrow \lambda(x{:}|\tau|).\, ()}\ \textbf{sub}\text{top}$$

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

$$\frac{\tau_1 <: [\alpha_1/\alpha_2]\tau_2 \hookrightarrow C}{\forall \alpha_1.\, \tau_1 <: \forall \alpha_2.\, \tau_2 \hookrightarrow \lambda(f{:}|\forall \alpha.\, \tau_1|).\, \Lambda \alpha.\, C\ (f\ \alpha)}\ \textbf{sub}\text{forall} \qquad \frac{\tau_1 <: \tau_2 \hookrightarrow C_1 \qquad \tau_1 <: \tau_3 \hookrightarrow C_2}{\tau_1 <: \tau_2 \,\&\, \tau_3 \hookrightarrow \lambda(x{:}|\tau_1|).\, (C_1\ x, C_2\ x)}\ \textbf{sub}\text{and}$$

$$\frac{\tau_1 <: \tau_3 \hookrightarrow C}{\tau_1 \,\&\, \tau_2 <: \tau_3 \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\, C\ (\mathtt{proj}_1 x)}\ \textbf{sub}\text{and}_1 \qquad \frac{\tau_2 <: \tau_3 \hookrightarrow C}{\tau_1 \,\&\, \tau_2 <: \tau_3 \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\, C\ (\mathtt{proj}_2 x)}\ \textbf{sub}\text{and}_2$$

$$\frac{\tau_1 <: \tau_2 \hookrightarrow C}{\{l{:}\tau_1\} <: \{l{:}\tau_2\} \hookrightarrow \lambda(x{:}|\{l{:}\tau_1\}|).\, C\ x}\ \textbf{sub}\text{rec}$$

Figure 6. Coercive subtyping.

$$\boxed{\gamma \vdash e : \tau \hookrightarrow E} \qquad \frac{(x, \tau) \in \gamma}{\gamma \vdash x : \tau \hookrightarrow x}\ \textbf{E}\text{var} \qquad \frac{}{\gamma \vdash \top : \top \hookrightarrow ()}\ \textbf{E}\text{top} \qquad \frac{\gamma, x{:}\tau \vdash e : \tau_1 \hookrightarrow E \qquad \gamma \vdash \tau}{\gamma \vdash \lambda(x{:}\tau).\, e : \tau \to \tau_1 \hookrightarrow \lambda(x{:}|\tau|).\, E}\ \textbf{E}\text{lam}$$

$$\frac{\gamma \vdash e_1 : \tau_1 \to \tau_2 \hookrightarrow E_1 \qquad \gamma \vdash e_2 : \tau_3 \hookrightarrow E_2 \qquad \tau_3 <: \tau_1 \hookrightarrow C}{\gamma \vdash e_1\ e_2 : \tau_2 \hookrightarrow E_1\ (C\ E_2)}\ \textbf{E}\text{app} \qquad \frac{\gamma, \alpha \vdash e : \tau \hookrightarrow E}{\gamma \vdash \Lambda \alpha.\, e : \forall \alpha.\, \tau \hookrightarrow \Lambda \alpha.\, E}\ \textbf{E}\text{blam}$$

$$\frac{\gamma \vdash e : \forall \alpha.\, \tau_1 \hookrightarrow E \qquad \gamma \vdash \tau}{\gamma \vdash e\ \tau : [\tau/\alpha]\tau_1 \hookrightarrow E\ |\tau|}\ \textbf{E}\text{tapp} \qquad \frac{\gamma \vdash e_1 : \tau_1 \hookrightarrow E_1 \qquad \gamma \vdash e_2 : \tau_2 \hookrightarrow E_2}{\gamma \vdash e_1, , e_2 : \tau_1 \,\&\, \tau_2 \hookrightarrow (E_1, E_2)}\ \textbf{E}\text{merge} \qquad \frac{\gamma \vdash e : \tau \hookrightarrow E}{\gamma \vdash \{l = e\} : \{l{:}\tau\} \hookrightarrow E}\ \textbf{E}\text{rec-construct}$$

$$\frac{\gamma \vdash e : \tau \hookrightarrow E \qquad \tau \bullet l = \tau_1 \hookrightarrow C}{\gamma \vdash e.l : \tau_1 \hookrightarrow C\ E}\ \textbf{E}\text{rec-select} \qquad \frac{\gamma \vdash e : \tau \hookrightarrow E \qquad \tau \setminus l = \tau_1 \hookrightarrow C}{\gamma \vdash e \setminus l : \tau_1 \hookrightarrow C\ E}\ \textbf{E}\text{rec-restrict}$$

$$\boxed{\tau_1 \bullet l = \tau_2 \hookrightarrow C} \qquad \frac{}{\{l{:}\tau\} \bullet l = \tau \hookrightarrow \lambda(x{:}|\{l{:}\tau\}|).\, x}\ \textbf{select} \qquad \frac{\tau_1 \bullet l = \tau \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \bullet l = \tau \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\, C\ (\mathtt{proj}_1 x)}\ \textbf{select}_1$$

$$\frac{\tau_2 \bullet l = \tau \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \bullet l = \tau \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\, C\ (\mathtt{proj}_2 x)}\ \textbf{select}_2$$

$$\boxed{\tau_1 \setminus l = \tau_2 \hookrightarrow C} \qquad \frac{}{\{l{:}\tau\} \setminus l = \top \hookrightarrow \lambda(x{:}|\{l{:}\tau\}|).\, ()}\ \textbf{restrict}$$

$$\frac{\tau_1 \setminus l = \tau \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \setminus l = \tau \,\&\, \tau_2 \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\, (C\ (\mathtt{proj}_1 x), \mathtt{proj}_2 x)}\ \textbf{restrict}_1$$

$$\frac{\tau_2 \setminus l = \tau \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \setminus l = \tau_1 \,\&\, \tau \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\, (\mathtt{proj}_1 x, C\ (\mathtt{proj}_2 x))}\ \textbf{restrict}_2$$

Figure 7. Elaboration typing from $F_{\&}$ to System F.

the coercion will keep the `name` field and replace the `age` field with a unit.

Lemma 5 (**restrict** rules produce type-correct coercion)**.** *If* $\tau \setminus l = \tau_1 \hookrightarrow C$*, then* $\epsilon \vdash C : |\tau| \to |\tau_1|$.

Proof. By structural induction of the derivation. □

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

Proof. (Sketch) By structural induction on the expression and the corresponding inference rule. The full proof can be found in the appendix. □

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, we have:

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

6. Implementation

We implemented the core functionalities of the $F_{\&}$ as part of a JVM-based compiler. The implementation supports record update instead of restriction as a primitive; however the former is formalized with the same underlying idea of elaborating records. Based on the type system of $F_{\&}$, we built an ML-like source language compiler that offers interoperability with Java (such as object creation and method calls). The source language is loosely based on the more general System F_ω (compared to our target, System F) 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 following phases in the compiler that 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 $F_{\&}$ expression 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 [38] 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 has also studied a system where both intersection types and bounded polymorphism are present in his Ph.D dissertation [39] and a 1997 report [40]. Going in the direction of higher kinds, Compagnoni and Pierce [12] add intersection types to System F_ω and use 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 [18] study the interactions between intersection types and effects in call-by-value languages. And they propose a ``value restriction'' for intersection types, similar to value restriction on parametric 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 have 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 3, 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. [14]. As emphasized throughout the paper our work is inspired by Dunfield [20]. He describes 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 [44] 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 unambigu-

ously. For instance, it is not allowed in Forsythe to merge two functions.

Refinement intersection [17, 19, 23] 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. [24] study 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* [21], 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* [22] 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, 31, 33, 45]. The work by Nystrom on *nested intersection* [33] uses a form of intersection types to support the composition of families of classes. Ostermann's *delegation layers* [36] 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 $F_{\&}$.

Extensible records. Encoding records using intersection types appear in Reynolds [44] and Castagna et al. [10]. Although Dunfield also discusses this idea in his paper [20], he only provides an implementation but not formalization. Very similar to our treatment of elaborating records is Cardelli's work [6] on translating a calculus, named $F_{<:\rho}$, with extensible records to a simpler calculus that without records primitives (in which case is $F_{<:}$). But he does not consider encoding multi-field records as intersections; hence his translation is more heavyweight. Crary [15] uses intersection types and existential types to address the problem that arises when interpreting method dispatch as self-application. But in his paper, intersection types are not used to encode multi-field records.

Wand [50] started the work on extensible records and proposes row types [51] for records. Cardelli and Mitchell [7] defined three primitive operations on records that are similar to ours: *selection*, *restriction*, and *extension*. The merge operator in $F_{\&}$ plays the same role as extension. Following Cardelli and Mitchell's approach, Leijen [29, 30] define record update in terms of restriction and extension. Both Leijen's system and ours allows records that contain duplicate labels. Arguably Leijen's system is stronger. For example, it supports passing record labels as arguments to functions. He also shows encoding an intersection types using first-class labels. bruno: check carefully this text! Chlipala's `Ur` [11] explains record as type level constructs.bruno: What is the point of citing Chlipala's paper?

8. Conclusion and Further Work

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

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

References

[1] N. Amin, A. Moors, and M. Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, number EPFL-CONF-183030, 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*, pages 233--249. ACM, 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, pages 183--200, 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*, FPCA '89, pages 273--280, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. . URL `http://doi.acm.org/10.1145/99370.99392`.

[6] L. Cardelli. *Extensible records in a pure calculus of subtyping*. Digital. Systems Research Center, 1992.

[7] L. Cardelli and J. C. Mitchell. Operations on records. In *Mathematical foundations of programming semantics*, pages 22--52. Springer, 1990.

[8] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471--523, 1985.

[9] L. Cardelli, S. Martini, J. Mitchell, and A. Scedrov. An extension of System F with subtyping. *Information and Computation*, 109:4--56, 1994. Preliminary version appeared *Proc. Theor. Aspects of Computer Software,* Springer LNCS 526, September 1991, pages 750--770.

[10] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115--135, 1995.

[11] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *ACM Sigplan Notices*, volume 45, pages 122--133. ACM, 2010.

[12] A. B. Compagnoni and B. C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469--501, 1996.

[13] 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*, pages 125--135. ACM, 1989.

[14] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45--58, 1981.

[15] K. Crary. Simple, efficient object encoding using intersection types. Technical report, Cornell University, 1998.

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

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

[18] R. Davies and F. Pfenning. Intersection types and computational effects. In *ACM Sigplan Notices*, volume 35, pages 198--208. ACM, 2000.

[19] J. Dunfield. Refined typechecking with stardust. In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 21--32. ACM, 2007.

[20] J. Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133--165, 2014.

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

[22] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. *POPL 2006*, pages 270--282.

[23] T. Freeman and F. Pfenning. *Refinement types for ML*, volume 26. ACM, 1991.

[24] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)*, 55(4):19, 2008.

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

[26] 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, pages 411--428, 1993.

[27] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 28(5):795--847, Sept. 2006.

[28] D. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2:127--145, 1968.

[29] D. Leijen. First-class labels for extensible rows. *UU-CS*, (2004-051), 2004.

[30] D. Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 5:297--312, 2005.

[31] 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.

[32] 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, pages 423--438, 2008.

[33] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *ACM SIGPLAN Notices*, volume 41, pages 21--36. ACM, 2006.

[34] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses. In *ECOOP 2012--Object-Oriented Programming*, pages 2--27. Springer, 2012.

[35] 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*, pages 27--51. Springer, 2013.

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

[37] K. Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121--145, 2008.

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

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

[40] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(02):129--193, 1997.

[41] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP '97 --- Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*. Springer-Verlag, 1997.

[42] 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, pages 377--395, New York, NY, USA, 2014. ACM. ISBN

978-1-4503-2585-1. . URL http://doi.acm.org/10.1145/2660193.2660237.

[43] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408--423, London, UK, UK, 1974. Springer-Verlag. ISBN 3-540-06859-7. URL http://dl.acm.org/citation.cfm?id=647323.721503.

[44] J. C. Reynolds. *Design of the programming language Forsythe*. Springer, 1997.

[45] 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.

[46] 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, pages 107--119, 1999.

[47] M. Torgersen. The Expression Problem Revisited. In M. Odersky, editor, *Proc. of the 18th European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 123--143, Oslo (Norway), June 2004.

[48] 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, pages 1289--1296, 2004.

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

[50] M. Wand. Complete type inference for simple objects. In *LICS*, volume 87, pages 37--44, 1987.

[51] M. Wand. Type inference for record concatenation and multiple inheritance. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 92--97. IEEE, 1989.

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

A. Type Well-formedness

$ftv(\cdot)$ reads: ``the free type variable of''.

$\boxed{\gamma \vdash \tau}$

$$\frac{ftv(\tau) \in \gamma}{\gamma \vdash \tau}\ \textbf{E}\text{wf}$$

Figure 8. Type well-formedness in $\mathsf{F}_{\&}$.

$\boxed{\Gamma \vdash T}$

$$\frac{ftv(T) \in \Gamma}{\Gamma \vdash T}\ \textbf{T}\text{wf}$$

Figure 9. Type well-formedness in the target type system.

B. Target Type System

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

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T}\ \textbf{T}\text{var} \qquad \frac{}{\Gamma \vdash () : ()}\ \textbf{T}\text{unit} \qquad \frac{\Gamma, x{:}T \vdash E : T_1 \qquad \Gamma \vdash T}{\Gamma \vdash \lambda(x{:}T).\, E : T \to T_1}\ \textbf{T}\text{lam} \qquad \frac{\Gamma \vdash E_1 : T_1 \to T_2 \qquad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1\ E_2 : T_2}\ \textbf{T}\text{app}$$

$$\frac{\Gamma, \alpha \vdash E : T}{\Gamma \vdash \Lambda\alpha.\, E : \forall\alpha.\, T}\ \textbf{T}\text{blam} \qquad \frac{\Gamma \vdash E : \forall\alpha.\, T_1 \qquad \Gamma \vdash T}{\Gamma \vdash E\ T : [T/\alpha]T_1}\ \textbf{T}\text{tapp} \qquad \frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : (T_1, T_2)}\ \textbf{T}\text{pair}$$

$$\frac{\Gamma \vdash E : (T_1, T_2)}{\Gamma \vdash \mathtt{proj}_1 E : T_1}\ \textbf{T}\text{proj}_1 \qquad \frac{\Gamma \vdash E : (T_1, T_2)}{\Gamma \vdash \mathtt{proj}_2 E : T_2}\ \textbf{T}\text{proj}_2$$

Figure 10. Target type system.

C. Proofs

Notation. We sketch our proofs in two-column style: on the left are the intermediate results and on the right are the justification (for the previous intermediate result to reach the corresponding left-hand side).

C.1 Elaboration

bruno: fix numbering of lemmas bruno: reflexitivity and transitivity missing. You can do a proof sketch instead of a full proof. Just say in 1 or 2 sentences what is the main idea. You can mention that we have a full proof in Coq. bruno: target type system is missing 3 cases: Tunit; Tproj1; TProj2

Lemma 6 (sub rules produce type-correct coercion). *If* $\tau_1 <: \tau_2 \hookrightarrow C$, *then* $\epsilon \vdash C : |\tau_1| \to |\tau_2|$.

Proof. By structural induction of the derivation.

- **Case**

  $$\frac{}{\alpha <: \alpha \hookrightarrow \lambda(x{:}|\alpha|).\, x}\ \textbf{sub}\text{var}$$

  $\epsilon \vdash \lambda(x{:}|\alpha|).\, x : \alpha \to \alpha$ By **T**var and **T**lam

- **Case**

$$\frac{}{\tau <: \top \hookrightarrow \lambda(x{:}|\tau|).\ ()}\ \textbf{sub}\text{top}$$

	$\epsilon \vdash \lambda(x{:}\|\tau\|).\ () : \|\tau\| \to ()$	By **T**var and **T**lam
	$\epsilon \vdash \lambda(x{:}\|\tau\|).\ () : \|\tau\| \to \|\top\|$	By the definition of $\|\cdot\|$

- **Case**

$$\frac{\tau_3 <: \tau_1 \hookrightarrow C_1 \qquad \tau_2 <: \tau_4 \hookrightarrow C_2}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4 \hookrightarrow \lambda(f{:}|\tau_1 \to \tau_2|).\lambda(x{:}|\tau_3|).\ C_2\ (f\ (C_1\ x))}\ \textbf{sub}\text{fun}$$

	$\tau_3 <: \tau_1 \hookrightarrow C_1$	Premise
	$\epsilon \vdash C_1 : \|\tau_3\| \to \|\tau_1\|$	By i.h.
	$\epsilon \vdash C_2 : \|\tau_2\| \to \|\tau_4\|$	Similar to the above

george: TODO

- **Case**

$$\frac{\tau_1 <: [\alpha_1/\alpha_2]\tau_2 \hookrightarrow C}{\forall\alpha_1.\tau_1 <: \forall\alpha_2.\tau_2 \hookrightarrow \lambda(f{:}|\forall\alpha.\tau_1|).\Lambda\alpha.\ C\ (f\ \alpha)}\ \textbf{sub}\text{forall}$$

george: TODO

- **Case**

$$\frac{\tau_1 <: \tau_2 \hookrightarrow C_1 \qquad \tau_1 <: \tau_3 \hookrightarrow C_2}{\tau_1 <: \tau_2 \,\&\, \tau_3 \hookrightarrow \lambda(x{:}|\tau_1|).\ (C_1\ x, C_2\ x)}\ \textbf{sub}\text{and}$$

george: TODO

- **Case**

$$\frac{\tau_1 <: \tau_3 \hookrightarrow C}{\tau_1 \,\&\, \tau_2 <: \tau_3 \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\ C\ (\mathtt{proj}_1 x)}\ \textbf{sub}\text{and}_1$$

george: TODO

- **Case**

$$\frac{\tau_2 <: \tau_3 \hookrightarrow C}{\tau_1 \,\&\, \tau_2 <: \tau_3 \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\ C\ (\mathtt{proj}_2 x)}\ \textbf{sub}\text{and}_2$$

By symmetry with the above case.

- **Case**

$$\frac{\tau_1 <: \tau_2 \hookrightarrow C}{\{l{:}\tau_1\} <: \{l{:}\tau_2\} \hookrightarrow \lambda(x{:}|\{l{:}\tau_1\}|).\ C\ x}\ \textbf{sub}\text{rec}$$

	$\tau_1 <: \tau_2 \hookrightarrow C$	Premise
(a)	$\epsilon \vdash C : \|\tau_1\| \to \|\tau_2\|$	By i.h.
	$\epsilon, x{:}\|\{l{:}\tau_1\}\| \vdash x : \|\{l{:}\tau_1\}\|$	By **T**var
	$\epsilon, x{:}\|\{l{:}\tau_1\}\| \vdash x : \|\tau_1\|$	By the definition of $\|\cdot\|$
	$\epsilon, x{:}\|\{l{:}\tau_1\}\| \vdash C\ x : \|\tau_2\|$	By **T**app and (a)
	$\epsilon, x{:}\|\{l{:}\tau_1\}\| \vdash C\ x : \|\{l{:}\tau_2\}\|$	By the definition of $\|\cdot\|$
	$\epsilon \vdash \lambda(x{:}\|\{l{:}\tau_1\}\|).\ C\ x : \|\{l{:}\tau_1\}\| \to \|\{l{:}\tau_2\}\|$	By **T**lam

□

Lemma 7 (**select** rules produce type-correct coercion)**.** *If* $\tau \bullet \mathsf{l} = \tau_1 \hookrightarrow C$*, then* $\epsilon \vdash C : |\tau| \to |\tau_1|$.

Proof. By structural induction of the derivation.

- **Case**

$$\frac{}{\{\mathsf{l}{:}\tau\} \bullet \mathsf{l} = \tau \hookrightarrow \lambda(x{:}|\{\mathsf{l}{:}\tau\}|).\, x}\ \textbf{select}$$

$\epsilon \vdash \lambda(x{:}\|\{\mathsf{l}{:}\tau\}\|).\, x : \|\{\mathsf{l}{:}\tau\}\| \to \|\{\mathsf{l}{:}\tau\}\|$	By **T**lam and **T**var
$\epsilon \vdash \lambda(x{:}\|\{\mathsf{l}{:}\tau\}\|).\, x : \|\{\mathsf{l}{:}\tau\}\| \to \|\tau\|$	By the definition of $\|\cdot\|$

- **Case**

$$\frac{\tau_1 \bullet \mathsf{l} = \tau \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \bullet \mathsf{l} = \tau \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\, C\ (\mathtt{proj}_1\, x)}\ \textbf{select}_1$$

$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash x : \|\tau_1 \,\&\, \tau_2\|$	By **T**var
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash x : (\|\tau_1\|, \|\tau_2\|)$	By the definition of $\|\cdot\|$
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash \mathtt{proj}_1\, x : \|\tau_1\|$	By **T**proj$_1$
$\epsilon \vdash C : \|\tau_1\| \to \|\tau\|$	By i.h.
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash C : \|\tau_1\| \to \|\tau\|$	By weakening
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash C\ (\mathtt{proj}_1\, x) : \|\tau\|$	By **T**app
$\epsilon \vdash \lambda(x{:}\|\tau_1 \,\&\, \tau_2\|).\, C\ (\mathtt{proj}_1\, x) : \|\tau_1 \,\&\, \tau_2\| \to \|\tau\|$	By **T**lam

- **Case**

$$\frac{\tau_2 \bullet \mathsf{l} = \tau \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \bullet \mathsf{l} = \tau \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\, C\ (\mathtt{proj}_2\, x)}\ \textbf{select}_2$$

By symmetry with the above case.

□

Lemma 8 (**restrict** rules produce type-correct coercion)**.** *If* $\tau \setminus \mathsf{l} = \tau_1 \hookrightarrow C$*, then* $\epsilon \vdash C : |\tau| \to |\tau_1|$.

Proof. By structural induction of the derivation.

- **Case**

$$\frac{}{\{\mathsf{l}{:}\tau\} \setminus \mathsf{l} = \top \hookrightarrow \lambda(x{:}|\{\mathsf{l}{:}\tau\}|).\, ()}\ \textbf{restrict}$$

$\epsilon \vdash \lambda(x{:}\|\{\mathsf{l}{:}\tau\}\|).\, () : \|\{\mathsf{l}{:}\tau\}\| \to ()$	By **T**unit and **T**lam
$\epsilon \vdash \lambda(x{:}\|\{\mathsf{l}{:}\tau\}\|).\, () : \|\{\mathsf{l}{:}\tau\}\| \to \|\top\|$	By the definition of $\|\cdot\|$

- **Case**

$$\frac{\tau_1 \setminus \mathsf{l} = \tau \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \setminus \mathsf{l} = \tau \,\&\, \tau_2 \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\, (C\ (\mathtt{proj}_1\, x), \mathtt{proj}_2\, x)}\ \textbf{restrict}_1$$

$\tau_1 \setminus \mathsf{l} = \tau \hookrightarrow C$	Premise
$\epsilon \vdash C : \|\tau_1\| \to \|\tau\|$	By i.h.
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash x : \|\tau_1 \,\&\, \tau_2\|$	By **T**var
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash x : (\|\tau_1\|, \|\tau_2\|)$	By the definition of $\|\cdot\|$
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash \mathtt{proj}_1\, x : \|\tau_1\|$	By **T**proj$_1$
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash \mathtt{proj}_2\, x : \|\tau_2\|$	By **T**proj$_2$
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash C\ (\mathtt{proj}_1\, x) : \|\tau\|$	By **T**app
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash (C\ (\mathtt{proj}_1\, x), \mathtt{proj}_2\, x) : (\|\tau\|, \|\tau_2\|)$	By **T**pair
$\epsilon, x{:}\|\tau_1 \,\&\, \tau_2\| \vdash (C\ (\mathtt{proj}_1\, x), \mathtt{proj}_2\, x) : \|\tau \,\&\, \tau_2\|$	By the definition of $\|\cdot\|$
$\epsilon \vdash \lambda(x{:}\|\tau_1 \,\&\, \tau_2\|).\, (C\ (\mathtt{proj}_1\, x), \mathtt{proj}_2\, x) : \|\tau_1 \,\&\, \tau_2\| \to \|\tau \,\&\, \tau_2\|$	By **T**lam

- **Case**

$$\frac{\tau_2 \setminus l = \tau \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \setminus l = \tau_1 \,\&\, \tau \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\,(\mathtt{proj}_1 x, C\ (\mathtt{proj}_2 x))}\ \mathtt{restrict}_2$$

  By symmetry with the above case.

□

Lemma 9 (**update** rules produce type-correct coercion)**.** *If* $\tau \blacktriangleleft \{l{:}\tau_1 \hookrightarrow E\} = \tau_2\lfloor\tau_3\rfloor \hookrightarrow C$ *and* $\Gamma \vdash E : |\tau_1|$ *for some* Γ*, then* $\Gamma \vdash C : |\tau| \to |\tau_2|$.

Proof. By structural induction of the derivation.

- **Case**

$$\frac{}{\{l{:}\tau\} \blacktriangleleft \{l{:}\tau_1 \hookrightarrow E\} = \{l{:}\tau_1\}\lfloor\tau\rfloor \hookrightarrow \lambda(\_{:}|\{l{:}\tau\}|).\,E}\ \mathtt{update}$$

  $\Gamma \vdash \lambda(\_{:}|\{l{:}\tau\}|).\,E : |\{l{:}\tau\}| \to |\tau_1|$ By Tlam, Tvar, and the hypothesis

- **Case**

$$\frac{\tau_1 \blacktriangleleft \{l{:}\tau \hookrightarrow E\} = \tau_3\lfloor\tau_4\rfloor \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \blacktriangleleft \{l{:}\tau \hookrightarrow E\} = \tau_3 \,\&\, \tau_2\lfloor\tau_4\rfloor \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\,C\ (\mathtt{proj}_1 x)}\ \mathtt{update}_1$$

$\Gamma, x{:}\lvert\tau_1 \,\&\, \tau_2\rvert \vdash x : \lvert\tau_1 \,\&\, \tau_2\rvert$	By Tvar
$\Gamma, x{:}\lvert\tau_1 \,\&\, \tau_2\rvert \vdash x : (\lvert\tau_1\rvert, \lvert\tau_2\rvert)$	By the definition of $\lvert\cdot\rvert$
$\Gamma, x{:}\lvert\tau_1 \,\&\, \tau_2\rvert \vdash \mathtt{proj}_1 x : \lvert\tau_1\rvert$	By $\mathrm{T}\mathtt{proj}_1$
$\Gamma \vdash C : \lvert\tau_1\rvert \to \lvert\tau_3\rvert$	By i.h.
$\Gamma, x{:}\lvert\tau_1 \,\&\, \tau_2\rvert \vdash C : \lvert\tau_1\rvert \to \lvert\tau_3\rvert$	By weakening
$\Gamma, x{:}\lvert\tau_1 \,\&\, \tau_2\rvert \vdash C\ (\mathtt{proj}_1 x) : \lvert\tau_3\rvert$	By Tapp
$\Gamma \vdash \lambda(x{:}\lvert\tau_1 \,\&\, \tau_2\rvert).\,C\ (\mathtt{proj}_1 x) : \lvert\tau_1 \,\&\, \tau_2\rvert \to \lvert\tau_3\rvert$	By Tlam

- **Case**

$$\frac{\tau_2 \blacktriangleleft \{l{:}\tau \hookrightarrow E\} = \tau_3\lfloor\tau_4\rfloor \hookrightarrow C}{\tau_1 \,\&\, \tau_2 \blacktriangleleft \{l{:}\tau \hookrightarrow E\} = \tau_1 \,\&\, \tau_3\lfloor\tau_4\rfloor \hookrightarrow \lambda(x{:}|\tau_1 \,\&\, \tau_2|).\,C\ (\mathtt{proj}_2 x)}\ \mathtt{update}_2$$

  By symmetry with the above case.

□

Lemma 10 (Preservation of well-formedness under type translation)**.** *If* $\gamma \vdash \tau$*, then* $|\gamma| \vdash |\tau|$.

Proof. By structural induction of the derivation. The only case to consider is Ewf.

- **Case**

$$\frac{\mathit{ftv}(\tau) \in \gamma}{\gamma \vdash \tau}\ \mathrm{Ewf}$$

$\mathit{ftv}(\tau) \in \gamma$	Premise
$\mathit{ftv}(\lvert\tau\rvert) \in \lvert\gamma\rvert$	By the definition of $\lvert\cdot\rvert$
$\lvert\gamma\rvert \vdash \lvert\tau\rvert$	By Twf

□

Theorem 3 (Translation preserves well-typing)**.** *If* $\gamma \vdash e : \tau \hookrightarrow E$*, then* $|\gamma| \vdash E : |\tau|$.

Proof. By structural induction of the derivation.

- **Case**

$$\frac{(x,\tau) \in \gamma}{\gamma \vdash x : \tau \hookrightarrow x}\ \textsc{Evar}$$

$(x,\tau) \in \gamma$	Premise
$(x,\|\tau\|) \in \|\gamma\|$	By the definition of $\|\cdot\|$
$\|\gamma\| \vdash x : \|\tau\|$	By Tvar

- **Case**

$$\frac{}{\gamma \vdash \top : \top \hookrightarrow ()}\ \textsc{Etop}$$

$\|\gamma\| \vdash () : ()$	By Tunit
$\|\gamma\| \vdash () : \|\top\|$	By the definition of $\|\cdot\|$

- **Case**

$$\frac{\gamma, x{:}\tau \vdash e : \tau_1 \hookrightarrow E \qquad \gamma \vdash \tau}{\gamma \vdash \lambda(x{:}\tau).\, e : \tau \to \tau_1 \hookrightarrow \lambda(x{:}|\tau|).\, E}\ \textsc{Elam}$$

$\gamma, x{:}\tau \vdash e : \tau_1 \hookrightarrow E$	Premise
$\|\gamma, x{:}\tau\| \vdash E : \|\tau_1\|$	By i.h.
$\|\gamma\|, x{:}\|\tau\| \vdash E : \|\tau_1\|$	By the definition of $\|\cdot\|$
$\|\gamma\| \vdash \lambda(x{:}\|\tau\|).\, E : \|\tau\| \to \|\tau_1\|$	By Tlam
$\|\gamma\| \vdash \lambda(x{:}\|\tau\|).\, E : \|\tau \to \tau_1\|$	By the definition of $\|\cdot\|$

- **Case**

$$\frac{\gamma \vdash e_1 : \tau_1 \to \tau_2 \hookrightarrow E_1 \qquad \gamma \vdash e_2 : \tau_3 \hookrightarrow E_2 \qquad \tau_3 <: \tau_1 \hookrightarrow C}{\gamma \vdash e_1\ e_2 : \tau_2 \hookrightarrow E_1\ (C\ E_2)}\ \textsc{Eapp}$$

$\gamma \vdash e_1 : \tau_1 \to \tau_2 \hookrightarrow E_1$	Premise
$\|\gamma\| \vdash E_1 : \|\tau_1 \to \tau_2\|$	By i.h.
$\gamma \vdash e_2 : \tau_3 \hookrightarrow E_2$	Premise
$\|\gamma\| \vdash E_2 : \|\tau_3\|$	By i.h.
$\tau_3 <: \tau_1 \hookrightarrow C$	Premise
$\epsilon \vdash C : \|\tau_3\| \to \|\tau_1\|$	By Lemma 6
$\|\gamma\| \vdash E_1\ (C\ E_2) : \|\tau_2\|$	By Tapp and the definition of $\|\cdot\|$

- **Case**

$$\frac{\gamma, \alpha \vdash e : \tau \hookrightarrow E}{\gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, \tau \hookrightarrow \Lambda\alpha.\, E}\ \textsc{Eblam}$$

$\gamma, \alpha \vdash e : \tau \hookrightarrow E$	Premise
$\|\gamma, \alpha\| \vdash E : \|\tau\|$	By i.h.
$\|\gamma\|, \alpha \vdash E : \|\tau\|$	By the definition of $\|\cdot\|$
$\|\gamma\| \vdash \Lambda\alpha.\, E : \forall\alpha.\, \|\tau\|$	By Tblam
$\|\gamma\| \vdash \Lambda\alpha.\, E : \|\forall\alpha.\, \tau\|$	By the definition of $\|\cdot\|$

- **Case**

$$\frac{\gamma \vdash e : \forall\alpha.\, \tau_1 \hookrightarrow E \qquad \gamma \vdash \tau}{\gamma \vdash e\ \tau : [\tau/\alpha]\tau_1 \hookrightarrow E\ |\tau|}\ \textsc{Etapp}$$

$\gamma \vdash e : \forall\alpha.\,\tau_1 \hookrightarrow E$	Premise
$\lvert\gamma\rvert \vdash E : \lvert\forall\alpha.\,\tau_1\rvert$	By i.h.
$\lvert\gamma\rvert \vdash E : \forall\alpha.\,\lvert\tau_1\rvert$	By the definition of $\lvert\cdot\rvert$
$\gamma \vdash \tau$	Premise
$\lvert\gamma\rvert \vdash \lvert\tau\rvert$	By Lemma 10
$\gamma \vdash E\ \lvert\tau\rvert : [\lvert\tau\rvert/\alpha]\lvert\tau_1\rvert$	By Ttapp
$\gamma \vdash E\ \lvert\tau\rvert : \lvert[\tau/\alpha]\tau_1\rvert$	By substitution lemma

- **Case**

$$\frac{\gamma \vdash e_1 : \tau_1 \hookrightarrow E_1 \qquad \gamma \vdash e_2 : \tau_2 \hookrightarrow E_2}{\gamma \vdash e_1,, e_2 : \tau_1 \,\&\, \tau_2 \hookrightarrow (E_1, E_2)}\ \text{Emerge}$$

$\gamma \vdash e_1 : \tau_1 \hookrightarrow E_1$	Premise
$\lvert\gamma\rvert \vdash E_1 : \lvert\tau_1\rvert$	By i.h.
$\lvert\gamma\rvert \vdash E_2 : \lvert\tau_2\rvert$	Similar to the above
$\lvert\gamma\rvert \vdash (E_1, E_2) : (\lvert\tau_1\rvert, \lvert\tau_2\rvert)$	By Tpair
$\lvert\gamma\rvert \vdash (E_1, E_2) : \lvert\tau_1 \,\&\, \tau_2\rvert$	By the definition of $\lvert\cdot\rvert$

- **Case**

$$\frac{\gamma \vdash e : \tau \hookrightarrow E}{\gamma \vdash \{l = e\} : \{l{:}\tau\} \hookrightarrow E}\ \text{Erec-construct}$$

$\gamma \vdash e : \tau \hookrightarrow E$	Premise
$\lvert\gamma\rvert \vdash E : \lvert\tau\rvert$	By i.h.
$\lvert\gamma\rvert \vdash E : \lvert\{l{:}\tau\}\rvert$	By the definition of $\lvert\cdot\rvert$

- **Case**

$$\frac{\gamma \vdash e : \tau \hookrightarrow E \qquad \tau \bullet l = \tau_1 \hookrightarrow C}{\gamma \vdash e.l : \tau_1 \hookrightarrow C\ E}\ \text{Erec-select}$$

$\tau \bullet l = \tau_1 \hookrightarrow C$	Premise
$\epsilon \vdash C : \lvert\tau\rvert \to \lvert\tau_1\rvert$	By Lemma 7
$\lvert\gamma\rvert \vdash C : \lvert\tau\rvert \to \lvert\tau_1\rvert$	By weakening
$\gamma \vdash e : \tau \hookrightarrow E$	Premise
$\lvert\gamma\rvert \vdash E : \lvert\tau\rvert$	By i.h.
$\lvert\gamma\rvert \vdash C\ E : \lvert\tau_1\rvert$	By Tapp

- **Case**

$$\frac{\gamma \vdash e : \tau \hookrightarrow E \qquad \tau \setminus l = \tau_1 \hookrightarrow C}{\gamma \vdash e \setminus l : \tau_1 \hookrightarrow C\ E}\ \text{Erec-restrict}$$

$\tau \setminus l = \tau_1 \hookrightarrow C$	Premise
$\epsilon \vdash C : \lvert\tau\rvert \to \lvert\tau_1\rvert$	By Lemma 8
$\lvert\gamma\rvert \vdash C : \lvert\tau\rvert \to \lvert\tau_1\rvert$	By weakening
$\gamma \vdash e : \tau \hookrightarrow E$	Premise
$\lvert\gamma\rvert \vdash E : \lvert\tau\rvert$	By i.h.
$\lvert\gamma\rvert \vdash C\ E : \lvert\tau_1\rvert$	By Tapp

□