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* for improved *extensibility* of programs. Examples include *Object Algebras*, *Modular Visitors* or Torgersen's four design patterns using generics. Although those design patterns give practical benefits in terms of extensibility, they also expose limitations in existing mainstream OOP languages. In particular two pressing limitations are: 1) the lack of good mechanisms for *object-level* composition; 2) the *conflation of (type) inheritance with subtyping*.

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 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 to allow dynamic composition of values/objects. Moreover, in System $F_{\&}$ type inheritance is independent of subtyping, and it is possible for an extension to be a supertype of a base object type. System $F_{\&}$ is formalized and implemented. Furthermore the paper illustrates how various extensible designs can be encoded in System $F_{\&}$.

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

General Terms Design, Languages, Theory

Keywords Intersecion Types, Polymorphism, Type System

bruno: Make all figures fit and be well-formated!

1. Introduction

bruno: The following are just some loose notes for the paper.

There has been a remarkable number of works aimed at improving support for extensibility in programming languages. These works include: visions of new programming models[]; new programming languages or language extensions[], and *design patterns* that can be used with existing mainstream languages.

Some of the more recent work on extensibility includes various proposals for design patterns. Examples include *Object Algebras* [], *Modular Visitors* [] or Torgersen's [] four design patterns using generics. In those approaches the idea is to use some advanced (but already available) features, such as generics, 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 those design patterns give practical benefits in terms of extensibility, they also expose limitations in existing mainstream OOP languages. One limitation is that existing OOP languages lack good mechanisms for *object-level* composition. As illustrated by Oliveira et al. [], highly modular programs using Object Algebras are best expressed using a form of *type-safe*, *dynamic*, *delegation*-based composition. Although such form of composition can be encoded in Scala, it requires the use of low-level reflection techniques, such as dynamic proxies, reflection or other forms of metaprogramming []. Moreover, the vast majority of modern OOP languages follows a model where

2) the *conflation of (type) extension with subtyping*.

The use of existing languages has shown that we can get quite far, but it has also exposed some limitations. For example, work on object algebras has shown that a delegation-based OO model would be better suited for OAs. Moreover, Object Algebra composition can benefit from intersection types. Additionally Object Algebras suffer from the "Inheritance is not Subtyping" problem [**?**]. Although there are very few examples in the literature illustrating programs where the separating inheritance from subtyping is desirable, object algebras illustrate a particularly compelling example of this.

Motivated by the insights gained in previous work, this paper presents a minimal core calculus that addresses current limitations and provides a better foundational model for statically typed delegation-based OOP? We show that Object Algebras fit nicely in this model.

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

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

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 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 to allow 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. System $F_{\&}$ is formalized and implemented. Furthermore the paper illustrates how various extensible designs can be encoded in System $F_{\&}$.

We present a polymorphic calculus containing intersection types and records, and show how this language can be used to solve various common tasks in functional programming in a nicer way.

Intersection types provides a power mechanism for functional programming, in particular for extensibility and allowing new forms of composition.

Prototype-based programming is one of the two major styles of object-oriented programming, the other being class-based programming which is featured in languages such as Java and C#. It has gained increasing popularity recently with the prominence of JavaScript in web applications. Prototype-based programming supports highly dynamic behaviors at run time that are not possible with traditional class-based programming. However, despite its flexibility, prototype-based programming is often criticized over concerns of correctness and safety. Furthermore, almost all prototype-based systems rely on the fact that the language is dynamically typed and interpreted.

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*.
- **Implementation and Examples:** An implementation of an extension of System $F_{\&}$, as well as the examples presented in the paper, are publicly available.

1.1 Other Notes

finitary overloading: yes but have other merits of intersection been explored?

– Compare Scala: – merge[A,B] = new A with B

– type IEval = eval : Int – type IPrint = print : String

– F[_]

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

george: Change the examples later to something very simple.

This section provides the reader with necessary intuition of $F_{\&}$. In $F_{\&}$, the central addition to the type system of System F is intersection types. A number of OO languages, such as Java, Scala, and Ceylongeorge: cite?, already support intersection types to different degrees. A common limitation in those languages, though, is that there is no introduction construct at the term level for intersection types. In contrast, there are introduction construct for function types (lambdas) and universal quantification (big lambdas) in most core calculi. To fill this gap, we allow intersecting any two terms at run time using a *merge* operator, similar to Dunfield's [13] approach. The key constructs are the "merge" operator, denoted by ,, at the term level and the corresponding type intersection operator, denoted by & at the type level.

The addition of intersection types to System F has a number of consequences, which we will explore one by one in the following subsections.

2.1 Intersection Types

We motivate the use of intersection types with overloaded function, as intersection types provide a simple mechanism for ad-hoc polymorphism, similar to what type classes in Haskell achieve. 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 return its string representation. In other words, it is also *both* a function from integers to strings as well as a function from boolean to strings. Therefore, in $F_{\&}$ it should be of following type:

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

Assuming we have the following two functions available,

```
let showInt : Int -> String = ...
let showBool : Bool -> String = ...
```

We may define `show` by merging the them 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).

2.2 Subtyping

As a result of intersection types, the type system of $F_{\&}$ permits a subtyping relation naturally. The subtyping also arises

from contravariant paramter types and covariant return types for functions. Subtyping in $F_{\&}$ is structual. We will explore the usefulness of such a type system in practice by showing various examples.

2.3 Intersection Types and Records

With intersection types, we are able to regard multi-field records as just merges of single-field records, for example:

```
{ x = 1, y = 2 }
```

is just syntatic sugar for

```
{ x = 1 } ,, { y = 2 }
```

In addition, a record is just a normal type and can be merged with any other terms, for example, this is a also valid expression

```
1 ,, { x = 2 }
```

and we can further extract a field out of it, such as

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

That is because a record type of the form $\{l : \tau\}$ can be thought as a normal type `t` tagged by the label `l`. In consequence, $F_{\&}$ supports a generalization of record elimination and update that works for any type. Functional update can be done using the *with* keyword:

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

2.4 Intersection Types and Parametric Polymorphism

The presence of both parametric polymorphism and intersection is critical, as we shall see in the next section, in solving modularity problems. The most simple example is a function that just uses the merge operator:

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

The key novel feature is that in our language we allow the intersection of type parameters.

3. Application

This section shows that the System F plus intersection types are enough for encoding extensible designs, and even beat the designs in languages with a much more sophisticated type system. In particular, $F_{\&}$ has two main advantages over existing languages:

1. It supports dynamic composition of intersecting values.
2. It supports contravariant parameter types in the subtyping relation.

Various solutions have been proposed to deal with the extensibility problems and many rely on heavyweight language features such as abstract methods and classes in Java.

bruno: I would like to see a story about Church Encodings in $F_{\&}$. Can you look at Pierce's papers and try to write something along those lines? That will be a good intro for object algebras and visitors!

These two features can be used to improve existing designs of modular programs.

The expression problem refers to the difficulty of adding a new operations and a new data variant without changing or duplicating existing code.

There has been recently a lightweight solution to the expression problem that takes advantage of covariant return types in Java. We show that FI is able to solve the expression problem in the same spirit. The A)

```
trait Expr {
  def eval: Int
}

class Lit(n: Int) extends Expr {
  def eval: Int = n
}

class Add(n: Int) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

Dunfield [13] notes that using merges as a mechanism of overloading is not as powerful as type classes.

3.1 Encoding Bounded Polymorphism

As $F_{\&}$ extends System F with intersection types, $F_{<:}$ extends System F with bounded polymorphism. $F_{<:}$ [**?**] allows giving an upper bound to the type variable in type abstractions. The idea of bounded universal quantification was discussed in the seminal paper by Cardelli and Wegner [**?**]. They show that bounded quantifiers are useful because it is able to solve the "loss of information" problem.

In fact, the extension of System F in the other direction, i.e., 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, programmers would expect that passing the user to the function is allowed. They are correct. Note that in the source language multi-field records are just syntatic sugars for merges of single-field records. Therefore, user is of a subtype of the parameter due to subtyping introduced by intersection types. So far so good. But there is a problem: what if programmers wants to access the `admin` field later, like:

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

They cannot do so as the above will not typecheck. After going through the function, the user now has only the type

```
{name: String}
```

This is rather undesired because it indeed has an `admin` field!

Bounded polymorphism enable the function to return the exact type of the argument so that we have no problem in accessing the `admin` field later. Consider the example below:

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

We do not have bounded polymorphism in the source language. But we can encode that via intersection types:

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

3.2 Object Algebras

Object algebras provide an alternative to *algebraic data types* (ADT).bruno: We are targeting an OO crowd. Mentioning algebraic datatypes is not going to be very useful there. For example, the following Haskell definition of the type of simple expressions

```
data Exp where
  Lit :: Int -> Exp
  Add :: Exp -> Exp -> Exp
```

can be expressed by the *interface* of an object algebra of simple expressions:

```
trait ExpAlg[E] {
  def lit(x: Int): E
  def add(e1: E, e2: E): E
}
```

Similar to ADT, data constructors in object algebras are represented by functions such as `lit` and `add` inside an interface `ExpAlg`. Different with ADT, the type of the expression itself is abtracted by a type parameter `E`.

which can be expressed similarly in $F_{\&}$ as:

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

Scala supports intersection types via the `with` keyword. The type `A with B` expresses the combined interface of `A` and `B`. The idea is similar to

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

in Java. [1]

The value level counterpart are functions of the type `A => B => A with B`. [2]

Our type system is a simple extension of System F; yet surprisingly, it is able to solve the limitations of using object algebras in languages such as Java and Scala. We will illustrate this point with an step-by-step of solving the expression problem using a source language built on top of $F_{\&}$.

[1] However, Java would require the `A` and `B` to be concrete types, whereas in Scala, there is no such restriction.

[2] FIXME

Oliveira noted that composition of object algebras can be cumbersome and intersection types provides a solution to that problem.

We first define an interface that supports the evaluation operation:

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

The interface is just a type synonym `IEval`. In $F_{\&}$, record types are structural and hence any value that satisfies this interface is of type `IEval` or of a subtype of `IEval`. [3]

In the following, `ExpAlg` is an object algebra interface of expressions with literal and addition case. And `evalAlg` is an object algebra for evluation of those expressions, which has type `ExpAlg Int`

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

Next, we define an interface that supports pretty printing.

```
type IPrint = { print : String };
let printAlg = {
  lit = \(x : Int). { print = x.toString() },
  add = \(x : IPrint). \(y : IPrint). { print = x.
      print.concat(" + ").concat(y.print) },
  sub = \(x : IPrint). \(y : IPrint). { print = x.
      print.concat(" - ").concat(y.print) }
};
```

Provided with the definitions above, we can then create values using the appropriate algebras. For example: defines two expressions.

The expressions are unusual in the sense that they are functions that take an extra argument `f`, the object algebras, and use the data constructors provided by the object algebra (factory) `f` such as `lit`, `add` and `sub` to create 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`.

```
let merge A B (f : ExpAlg A) (g : ExpAlg 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
};
```

If we would like to have an expression that supports both evaluation and pretty printing, we will need a mechanism to combine the evaluation and printing algebras. Intersection types allows such composition: the `merge` function, which

[3] Should be mentioned in S2.

takes two expression algebras to create a combined algebra. It does so by constructing a new expression algebra, a record whose each field is a function that delegates the input to the two algebras taken.

```
let newAlg = merge IEval IPrint subEvalAlg
    printAlg in
let o1 = e1 (IEval & IPrint) newAlg in
o1.print
```

Note that `o1` is a single object created that supports both evaluation and printing, thus achieving full feature-oriented programming.

3.3 Visitors

Constructing instances seems clumsy!

The visitor pattern allows adding new operations to existing structures without modifying those structures. The type of expressions are defined as follows:

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

trait SubExp[A] extends Exp[A] {
  override def accept(f: SubExpAlg[A]): A
}
```

The body of `Exp` and `SubExp` are almost the same: they both contain an `accept` method that takes an algebra `f` and returns a value of the carrier type `A`. The only difference is at `f` — `SubExpAlg[A]` is a subtype of `ExpAlg[A]`. Since `f` appear in parameter position of `accept` and function parameters are contravariant, naturally we would hope that `SubExp[A]` is a supertype of `Exp[A]`. However, such subtyping relation does not fit well in Scala because inheritance implies subtyping in such languages [4]. As `SubExp[A]` extends `Exp[A]`, the former becomes a subtype of the latter.

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

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

Then by the typing judgment it holds that `SubExp` is a supertype of `Exp`. This relation gives desired results. To give a concrete example:

A is called is the *interpretation*. It works for any interpretation you want.

First we define two data constructors for simple expressions:

```
let lit (n : Int): Exp A = {
  accept = /\A. \(f : ExpAlg A). f.lit n
};

let add (e1 : Exp) (e2 : Exp): Exp A = {
  accept = /\A. \(f : ExpAlg A).
            f.add (e1.accept A f) (e2.accept A f)
};
```

Suppose later we decide to augment the expressions with subtraction:

```
let sub (e1 : SubExp) (e2 : SubExp): SubExp A =
  { accept = /\A. \(f : SubExpAlg A).
              f.sub (e1.accept A f) (e2.accept A f)
           };
```

One big benefit of using the visitor pattern is that programmers is able to write in the same way that would do in Haskell. For example, `e2 = sub (lit 2)(lit 3)` defines an expression.

Another important property that does not exist in Scala is that programmer is able to pass `lit 2`, which is of type `Exp A` , to `sub`, which expects a `SubExp A` because of the subtyping relation we have. After all, it is known statically that `lit 2` can be passed into `sub` and nothing will go wrong.bruno: Subtyping needs to be much more emphasized! See Modular Visitor Components!

3.4 Mixins

Mixins are useful programming technique wildly adopted in dynamic programming languages such as JavaScript and Ruby. But obviously it is the programmers' responsbility to make sure that the mixin does not try to access methods or fields that are not present in the base class.

In Haskell, one is also able to write programs in mixin style using records. However, this approach has a serious drawback: since there is no subtyping in Haskell, it is not possible to refine the mixin by adding more fields to the records. This means that the type of the family of the mixins has to be determined upfront, which undermines extensibility.

$F_{\&}$ is able to overcome both of the problems: it allows composing mixins that (1) extends the base behavior, (2) while ensuring type safety.

The figure defines a mini mixin library. The apostrophe in front of types denotes call-by-name arguments similar to the `=>` notation in the Scala language.

```
type Mixin S = 'S -> 'S -> S;
let zero S (super : 'S) (this : 'S) : S = super;
let rec mixin S (f : Mixin S) : S
  = let m = mixin S in f (\ (_ : Unit). m f) (\ (_
      : Unit). m f);
let extends S (f : Mixin S) (g : Mixin S) : Mixin
    S
  = \ (super : 'S). \ (this : 'S). f (\ (d : Unit).
      g super this) this;
```

We define a factorial function in mixin style and make a `noisy` mixin that prints "Hello" and delegates to its superclass. Then the two functions are composed using the `mixin` and `extends` combinators. The result is the `noisyFact` function that prints "Hello" every time it is called and computes factorial.

```
let fact (super : 'Int -> Int) (this : 'Int -> Int
    ) : Int -> Int
```

[4] It is still possible to encode contravariant parameter types in Scala but doing so would require some technique.bruno: what technique?

Types

τ	$::=$	$\alpha \mid \tau \to \tau \mid \forall\alpha.\tau$	
	$\mid$	$\tau \,\&\, \tau$	Intersection type
	$\mid$	$\{l : \tau\}$	Record type

Terms

e	$::=$	x	
	$\mid$	$\Lambda\alpha.\, e \mid e\ \tau$	Type abstraction/application
	$\mid$	$\lambda(x\!:\!\tau).\, e \mid e\ e$	Term abstraction/application
	$\mid$	$e,, e$	Merge
	$\mid$	$\{l = e\}$	Record construction
	$\mid$	$e.l$	Record projection
	$\mid$	e `with` $\{l = e\}$	Record update

Contexts

$\Gamma \quad ::= \quad \epsilon \mid \Gamma, \alpha \mid \Gamma, x\!:\!\tau$

Labels

l

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

```
  = \ (n : Int). if n == 0 then 1 else n * this (n
      - 1)
let noisy (super : 'Int -> Int) (this : 'Int ->
    Int) : Int -> Int
  = \ (n : Int). { println("Hello"); super n }
let noisyFact = mixin (Int -> Int) (extends (Int
    -> Int) foolish fact)
noisy 5
```

4. The $F_{\&}$ calculus

Following Dunfield's [13] work on simply-typed lambda calculus with intersection and union types, we formalize the syntax, subtyping, and typing of $F_{\&}$. In the next section, we will go through the type-directed translation from $F_{\&}$ to System F.

4.1 Syntax

Figure 1 shows the syntax of $F_{\&}$. It is System F at its core. To System F, we add two features: intersection types and single-field records. We include only single records because single record types as the multi-records can be desugared into the merge of multiple single records.

Types. The constructs in the first row of types in Figure 1 are standard in System F: type variable α, function types $\tau \to \tau$, and type abstraction $\forall\alpha.\tau$. The last two are novel. τ_1 & τ_1 is the intersection of type τ_1 and τ_2, and $\{l : \tau\}$ are the types for single-field records.

Expressions. The first five constructs of expressions are also standard: variables x, and two abstraction-elimination pairs: lambda expressions $\lambda(x : \tau).\ e$ abstract expression e over values of type τ and are eliminated by application $e\ e$; Big lambdas $\Lambda\alpha.\ e$ abstract expression e over types and are eliminated by type application $e\ \tau$. In the source language, lambdas are written as george: TODO and big lambdas as george: TODO.

The last four constructs are new: $e_1,, e_2$ is the *merge* of two expressions e_1 and e_2. It can be used as either e_1 or e_2. Particarly, if one regard e_1 and e_2 as objects, their merge will responds to every method that one or both of them have. Merge of expressions correspond to intersection types $\tau_1 \,\&\, \tau_2$. $\{l = e\}$ constructs a single-field record. $e.l$ accesses the field labelled l in e. Note that e does not need to be a record type in this case. For example, although the merge of two records

$$x = \{l_1 = e_1\},, \{l_1 = e_2\}$$

is of an intersection type, $x.l_1$ still gives e_1. On the other hand, $x.l_3$ does not type check. Functional update e `with` $\{l = e_1\}$ a *new* record which is exactly the same as e except the field labelled l is updated to become $e1$.

Context. Context Γ is also standard. It maps variables to their types and keeps bound type variables.

Discussion. A natural question the reader might ask is that why we have excluded union types from the language. The answer is we found that intersection types alone are enough support extensible designs. To focus on the key features that make this language interesting, we also omit other common constructs. For example, fixpoints can be added in standard ways.

4.2 Subtyping

george: Explain the subst syntax.

george: However, have we forbidden the interplay of subtyping relations explicitly declared by programmers as seen in class-based OO languages?

bruno: Intersection types require a simple form of subtyping. The subtyping relation is reflexive and transitive. Our subtyping relation follows previous work. Need to cite! The main difference is the interaction with parametric polymorphism.

In some calculi such as System $F_{<:}$, the subtyping relation is orthogoal to other language features: those calculi are indifferent with how the subtyping relation is defined. In $F_{\&}$, we take a syntatic approach, that is, subtyping is due to solely of intersection and function types.

Intersection types introduce natural subtyping relations among types. For example, `Int` & `Bool` should be a subtype of `Int`, since the former can be viewed as either `Int` or `Bool`. To summarize, the subtyping rules are standard except for three points listed below:

1. τ_1 & τ_2 is a subtype of τ_3, if *either* τ_1 or τ_2 are subtypes of τ_3,

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

$$\alpha <: \alpha \quad \text{(SVar)}$$

$$\frac{\tau_3 <: \tau_1 \qquad \tau_2 <: \tau_4}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4} \quad \text{(SFun)}$$

$$\frac{\tau_1 <: [\alpha_2 \mapsto \alpha_1]\tau_2}{\forall \alpha_1.\tau_1 <: \forall \alpha_2.\tau_2} \quad \text{(SForall)}$$

$$\frac{\tau_1 <: \tau_3}{\tau_1 \,\&\, \tau_2 <: \tau_3} \quad \text{(SAnd1)}$$

$$\frac{\tau_2 <: \tau_3}{\tau_1 \,\&\, \tau_2 <: \tau_3} \quad \text{(SAnd2)}$$

$$\frac{\tau_1 <: \tau_2 \qquad \tau_1 <: \tau_3}{\tau_1 <: \tau_2 \,\&\, \tau_3} \quad \text{(SAnd3)}$$

$$\frac{\tau_1 <: \tau_2}{\{l : \tau_1\} <: \{l : \tau_2\}} \quad \text{(SRec)}$$

Figure 2. Subtyping

2. τ_1 is a subtype of $\tau_2 \,\&\, \tau_3$, if τ_1 is a subtype of both τ_2 and τ_3.
3. $\{l_1 : \tau_1\}$ is a subtype of $\{l_2 : \tau_2\}$, if l_1 and l_2 are identical and τ_1 is a subtype of τ_2.

The first point is captured by two rules (S-And1) and (S-And2), whereas the second point by (S-And3). Note that the last point means that record types are covariant in the type of the fields.

4.3 Typing

The typing judgment for $F_{\&}$ is of the form: $\Gamma \vdash e : \tau$. This judgment uses the context Γ. The rules for variables, abstraction, type abstraction, and type application are standard in System F. To cater to the subtyping relations and avoid having undeterministic rules, in (App), we additionally require the type of the argument be a subtype of the parameter. The rule for record construction is also standard. For record projection and update, we resort to the auxiliary "get" and "put" rules. The "get" judgment checks if a field l indeed exists in a type τ and fetches the type of the field if so. The "put" judgment is almost similar, except that it takes the intended update (both expression and type), and returns in addition a new type of the expression. In record updates, we allow refining the type of the field in question.

In particular we introduce a (T-Merge) rule that applies to *merge* constructs.

$\boxed{\Gamma \vdash t}$

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \quad \text{(WF-Var)}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \quad \text{(WF-Fun)}$$

$$\frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha.\tau} \quad \text{(WF-Forall)}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \,\&\, \tau_2} \quad \text{(WF-And)}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \{l : \tau\}} \quad \text{(WF-Rec)}$$

Figure 3. Well-formedness

5. Type-directed Translation to System F

In this section we define the semantics of $F_{\&}$ by means of a type-directed translation to a variant of System F extended with tuples. This translation removes the labels of records and turns intersections into products, much like Dunfield's elaboration. But our translation also deals with parametric polymorphism and records.

5.1 Informal Discussion

To help the reader have a high-level understanding of how the translation works, in this subsection we present the translation informally. Take the $F_{\&}$ expression for example:

```
{ eval = 4, print = "4" }.eval
```

First, multi-field record literals are desugared into merges of single-field record literals. Therefore

```
{ eval = 4, print = "4" }
```

becomes

```
{ eval = 4 } ,, { print = "4" }
```

Merges of two values are translated into just a pair of them by (Merge) and single-field record literals lose their field labels by (RecCon). Hence `{ eval = 4 } ,, { print = "4"}` becomes `(4, "4")`.

bruno: Don't abuse inlining of examples in the text!

Finally, $e1$ and $e2$ are both coerced by a projection function $(x : (Int, String)).\texttt{fst}\ x$. bruno: Show the source program, and the program that it gets translated to. Then explain how that translation works.

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

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{(Var)}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \tau_1 \qquad \Gamma \vdash \tau}{\Gamma \vdash \lambda(x{:}\tau).\, e : \tau \to \tau_1} \quad \text{(Abs)}$$

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\tau} \quad \text{(TAbs)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_3 \qquad \tau_3 <: \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad \text{(App)}$$

$$\frac{\Gamma \vdash e : \forall\alpha.\tau_1 \qquad \Gamma \vdash \tau}{\Gamma \vdash e\ \tau : [\alpha \mapsto \tau]\tau_1} \quad \text{(TApp)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1,, e_2 : \tau_1 \,\&\, \tau_2} \quad \text{(Merge)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{l = e\} : \{l : \tau\}} \quad \text{(RecCon)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash_{\text{get}} (\tau; l) : \tau_1}{\Gamma \vdash e.l : \tau_1} \quad \text{(RecProj)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : \tau_1 \quad \vdash_{\text{put}} (t; l; e_1 : \tau_1) : (\tau_2, \tau_3) \qquad \tau_1 <: \tau_2}{\Gamma \vdash e\ \texttt{with}\ \{l = e_1\} : \tau_3} \quad \text{(RecUpd)}$$

$\boxed{\vdash_{\text{get}} (\tau_1; l) : \tau_2}$

$$\vdash_{\text{get}} (\{l : \tau\}; l) : \tau \quad \text{(GetBase)}$$

$$\frac{\vdash_{\text{get}} (\tau_1; l) : \tau}{\vdash_{\text{get}} (\tau_1 \,\&\, \tau_2; l) : \tau} \quad \text{(GetLeft)}$$

$$\frac{\vdash_{\text{get}} (\tau_2; l) : \tau}{\vdash_{\text{get}} (\tau_1 \,\&\, \tau_2; l) : \tau} \quad \text{(GetRight)}$$

$\boxed{\vdash_{\text{put}} (\tau; l; e : \tau) : (\tau, \tau)}$

$$\vdash_{\text{put}} (\{l : \tau\}; l; e : \tau_1) : (\tau, \{l : \tau_1\}) \quad \text{(PutBase)}$$

$$\frac{\vdash_{\text{put}} (\tau_1; l; e : \tau) : (\tau_3, \tau_4)}{\vdash_{\text{put}} (\tau_1 \,\&\, \tau_2; l; e : \tau) : (\tau_3, \tau_4 \,\&\, \tau_2)} \quad \text{(PutLeft)}$$

$$\frac{\vdash_{\text{put}} (\tau_2; l; e : \tau) : (\tau_3, \tau_4)}{\vdash_{\text{put}} (\tau_1 \,\&\, \tau_2; l; e : \tau) : (\tau_3, \tau_1 \,\&\, \tau_4)} \quad \text{(PutRight)}$$

Figure 4. Typing

$\boxed{[\![\tau]\!] = T}$

$$\begin{array}{rcl} [\![\alpha]\!] & = & \alpha \\ [\![\tau_1]\!] \to [\![\tau_2]\!] & = & [\![\tau_1]\!] \to [\![\tau_2]\!] \\ [\![\forall\alpha.\tau]\!] & = & \forall\alpha.[\![\tau]\!] \\ [\![\tau_1 \,\&\, \tau_2]\!] & = & \langle [\![\tau_1]\!], [\![\tau_2]\!] \rangle \\ [\![\{l : \tau\}]\!] & = & [\![t]\!] \end{array}$$

Figure 5. Type translation

5.2 Target Language

The target language is System F extended with pairs. The syntax and typing is completely standard. The syntax of System F is as follows:

$$\begin{array}{lrcl} \text{Types} & T & ::= & \alpha \mid T \to T \mid \forall\alpha.T \mid \langle T, T \rangle \\ \text{Terms} & E, C & ::= & x \mid \lambda(x{:}T).\, E \mid \Lambda\alpha.\, E \mid E\ E \mid E\ T \\ & & & \mid \langle E, E \rangle \mid \texttt{fst}E \mid \texttt{snd}E \end{array}$$

while its semantics can be found in standard texts [**?**].

The main translation judgment is $\Gamma \vdash e : \tau \hookrightarrow E$ which states that with respect to the environment Γ, the $F_{\&}$ expression e is of a $F_{\&}$ type τ and its translation is a System F expression E.

We also define the type translation function $[\![\cdot]\!]$ from $F_{\&}$ types τ to System F types T.

The first three rules of the translation is standard. For the last two, the intersection of two types are translated into a product of them, and the label of record types are erased.

The translation consists of four sets of rules, which are explained below:

5.3 Subtyping (Coercion)

Talk about η-expansion.

The coercion judgment $\Gamma \vdash \tau_1 <: \tau_2 \hookrightarrow C$ extends the subtyping judgment with a coercion on the right hand side of $\hookrightarrow$. A coercion C is an expression in the target language and has type $\tau_1 \to \tau_2$, as proved by Lemma 3. It is read "In the environment Γ, τ_1 is a subtype of τ_2; and if any expression e has a type τ_1 that is a subtype of the type of τ_2, the elaborated e, when applied to the corresponding coercion C, has exactly type $[\![\tau]\!]_2$". For example, $\Gamma \vdash Int\&Bool <: Bool \hookrightarrow fst$, where fst is the projection of a tuple on the first element. The coercion judgment is only used in the (App) case. As (SFun) supports contravariant parameter type and covariant return type, the coercion of the parameter types and that of the return types are used to create a coercion for the function type. (SAnd1), (SAnd2), and (SAnd3) deal with intersection types. The first two are complementary to each other. Take (SAnd1) for example, if we know τ_1 is a subtype of τ_3 and C is a coercion from τ_1 to τ_3, then we can conclude that $\tau_1 \,\&\, \tau_2$ is also a subtype of τ_3 and the new coercion is a function that takes a value x of type $\tau_1 \,\&\, \tau_2$, project x on the first item, and apply C to it. (SAnd3) uses both of two coercions and constructs a pair. bruno: Give a couple of concrete examples

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

$$\alpha <: \alpha \hookrightarrow \lambda(x : [\![\alpha]\!]).\ x \tag{SVar}$$

$$\frac{\tau_3 <: \tau_1 \hookrightarrow C_1 \qquad \tau_2 <: \tau_4 \hookrightarrow C_2}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4 \hookrightarrow \lambda(f : [\![\tau_1 \to \tau_2]\!]).\ \lambda(x : [\![\tau_3]\!]).\ C_2\ (f\ (C_1\ x))} \tag{SFun}$$

$$\frac{\tau_1 <: [\alpha_2 \mapsto \alpha_1]\tau_2 \hookrightarrow C}{\forall \alpha_1.\tau_1 <: \forall \alpha_2.\tau_2 \hookrightarrow \lambda(f : [\![\forall \alpha.\tau_1]\!]).\ \Lambda\alpha.\ C\ (f\ \alpha)} \tag{SForall}$$

$$\frac{\tau_1 <: \tau_3 \hookrightarrow C}{\tau_1\ \&\ \tau_2 <: \tau_3 \hookrightarrow \lambda(x : [\![\tau_1\ \&\ \tau_2]\!]).\ C\ (\texttt{fst}\ x)} \tag{SAnd1}$$

$$\frac{\tau_2 <: \tau_3 \hookrightarrow C}{\tau_1\ \&\ \tau_2 <: \tau_3 \hookrightarrow \lambda(x : [\![\tau_1\ \&\ \tau_2]\!]).\ C\ (\texttt{snd}\ x)} \tag{SAnd2}$$

$$\frac{\tau_1 <: \tau_2 \hookrightarrow C_1 \qquad \tau_1 <: \tau_3 \hookrightarrow C_2}{\tau_1 <: \tau_2\ \&\ \tau_3 \hookrightarrow \lambda(x : [\![\tau_1]\!]).\ \langle C_1\ x, C_2\ x\rangle} \tag{SAnd3}$$

$$\frac{\tau_1 <: \tau_2 \hookrightarrow C}{\{l : \tau_1\} <: \{l : \tau_2\} \hookrightarrow \lambda(x : [\![\{l : \tau_1\}]\!]).\ C\ x} \tag{SRec}$$

Figure 6. Coercion

when explaining the rules.

5.4 Typing (Translation)

In this subsection we now present formally the translation rules that convert $F_{\&}$ expressions into System F ones. This set of rules essentially extends those in the previous section with the light-blue part for the translation.

Translation The elaboration 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 $[\![\tau]\!]$. It is also standard, except for the case of (App), in which a coercion from the inferred type of the argument, e_2 , to the expected type of the parameter, τ_1, is inserted before the argument; (Merge) translates merges into pairs. (RecCon) uses the same System F expression E for e as for $\{l = e\}$. And in (RecEim) and (RecUpd) the coercions generated by the "get" and "put" rules will be used to coece the main $F_{\&}$ expression.

(RecProj) typechecks e and use the "get" rule to return the type of the field τ_1 and the coecion C. The type of the whole expression is τ_1 and its translation of $C\ E$.

(RecUpd) is similar to (RecProj) in that it uses the auxiliary "put" rule. This rule typechecks e and e_1, and uses the "put" rule. Note that it allows refining of types by an e_1 that is of a subtype of τ_1', which is the type of the field l in e. The type of the updated expression then takes the type τ' returned by the "put" rule, while its translation is E, applied to the coercion generated by the "put" rule, C.

The two set of rules are explained below.

"get" Rules The "get" judgment deals spefically with record elimination and yields a coercion can be thought as a field accessor. For example:bruno: Still not showing the derivations!

$\Gamma \vdash_{\text{get}} (\{\texttt{eval} : \texttt{Int}\}, \texttt{eval}) : \{\texttt{eval} : \texttt{Int}\} \hookrightarrow \lambda(x : \texttt{Int}).\ x$

The lambda is the field accessor and when applied to a translated expression of type $\{\texttt{eval} : \texttt{Int}\}$, it is able to give the desired field. (GetBase) is the base case: the type of the field labelled l in a $\{l : \tau\}$ is just τ and the coercion is an identity function specialized to type $[\![\{l : \tau\}]\!]$ (GetLeft) and (GetRight) are complementary to each other.

Consider the source program:

```
({ name = "Isaac", age = 10 }).name
```

Multi-field records are desugared into merge of single-field records:

```
({ name = "Isaac"} ,, { age = 10 }).name
```

By (GetBase),

$$\vdash_{\text{get}} (\{name : String\}; name) : String$$

we have the coercion

$$\lambda(x : [\![\{name : String\}]\!]).\ x$$

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

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau \hookrightarrow x} \quad \text{(Var)}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \tau_1 \hookrightarrow E \qquad \Gamma \vdash \tau}{\Gamma \vdash \lambda(x{:}\tau).\, e : \tau \to \tau_1 \hookrightarrow \lambda(x{:}[\![\tau]\!]).\, E} \quad \text{(Abs)}$$

$$\frac{\Gamma, \alpha \vdash e : \tau \hookrightarrow E}{\Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\tau \hookrightarrow \Lambda\alpha.\, E} \quad \text{(TAbs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \to \tau_2 \hookrightarrow E_1 \\ \Gamma \vdash e_2 : \tau_3 \hookrightarrow E_2 \qquad \tau_3 <: \tau_1 \hookrightarrow C\end{array}}{\Gamma \vdash e_1\ e_2 : \tau_2 \hookrightarrow E_1\ (C\ E_2)} \quad \text{(App)}$$

$$\frac{\Gamma \vdash e : \forall\alpha.\tau_1 \hookrightarrow E \qquad \Gamma \vdash \tau}{\Gamma \vdash e\ \tau : [\alpha \mapsto \tau]\tau_1 \hookrightarrow E\ [\![\tau]\!]} \quad \text{(TApp)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \hookrightarrow E_1 \qquad \Gamma \vdash e_2 : \tau_2 \hookrightarrow E_2}{\Gamma \vdash e_1,, e_2 : \tau_1 \,\&\, \tau_2 \hookrightarrow \langle E_1, E_2 \rangle} \quad \text{(Merge)}$$

$$\frac{\Gamma \vdash e : \tau \hookrightarrow E}{\Gamma \vdash \{l = e\} : \{l : \tau\} \hookrightarrow E} \quad \text{(RecCon)}$$

$$\frac{\Gamma \vdash e : \tau \hookrightarrow E \qquad \Gamma \vdash_{\text{get}} (\tau; l) : \tau_1 \hookrightarrow C}{\Gamma \vdash e.l : \tau_1 \hookrightarrow C\ E} \quad \text{(RecProj)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \hookrightarrow E \qquad \Gamma \vdash e_1 : \tau_1 \hookrightarrow E_1 \\ \vdash_{\text{put}} (t; l; e_1 : \tau_1 \hookrightarrow E_1) : (\tau_2, \tau_3) \hookrightarrow C \qquad \tau_1 <: \tau_2\end{array}}{\Gamma \vdash e\ \texttt{with}\ \{l = e_1\} : \tau_3 \hookrightarrow C\ E} \quad \text{(RecUpd)}$$

$\boxed{\vdash_{\text{get}} (\tau_1; l) : \tau_2 \hookrightarrow C}$

$$\vdash_{\text{get}} (\{l : \tau\}; l) : \tau \hookrightarrow \lambda(x{:}[\![\{l : \tau\}]\!]).\, x \quad \text{(GetBase)}$$

$$\frac{\vdash_{\text{get}} (\tau_1; l) : \tau \hookrightarrow C}{\vdash_{\text{get}} (\tau_1 \,\&\, \tau_2; l) : \tau \hookrightarrow \lambda(x{:}[\![\tau_1 \,\&\, \tau_2]\!]).\, C(\texttt{fst}\ x)} \quad \text{(GetLeft)}$$

$$\frac{\vdash_{\text{get}} (\tau_2; l) : \tau \hookrightarrow C}{\vdash_{\text{get}} (\tau_1 \,\&\, \tau_2; l) : \tau \hookrightarrow \lambda(x{:}[\![\tau_1 \,\&\, \tau_2]\!]).\, C(\texttt{snd}\ x)} \quad \text{(GetRight)}$$

$\boxed{\vdash_{\text{put}} (\tau; l; e : \tau \hookrightarrow E) : (\tau, \tau) \hookrightarrow C}$

$$\vdash_{\text{put}} (\{l : \tau\}; l; e : \tau_1 \hookrightarrow E) : (\tau, \{l : \tau_1\}) \hookrightarrow \lambda(x{:}[\![\{l : \tau\}]\!]).\, E \quad \text{(PutBase)}$$

$$\frac{\vdash_{\text{put}} (\tau_1; l; e : \tau \hookrightarrow E) : (\tau_3, \tau_4) \hookrightarrow C}{\vdash_{\text{put}} (\tau_1 \,\&\, \tau_2; l; e : \tau \hookrightarrow E) : (\tau_3, \tau_4 \,\&\, \tau_2) \hookrightarrow \lambda(x{:}[\![\tau_1 \,\&\, \tau_2]\!]).\, C(\texttt{fst}\ x)} \quad \text{(PutLeft)}$$

$$\frac{\vdash_{\text{put}} (\tau_2; l; e : \tau \hookrightarrow E) : (\tau_3, \tau_4) \hookrightarrow C}{\vdash_{\text{put}} (\tau_1 \,\&\, \tau_2; l; e : \tau \hookrightarrow E) : (\tau_3, \tau_1 \,\&\, \tau_4) \hookrightarrow \lambda(x{:}[\![\tau_1 \,\&\, \tau_2]\!]).\, C(\texttt{snd}\ x)} \quad \text{(PutRight)}$$

Figure 7. Type-directed translation from $F_{\&}$ to System F.

which is just $\lambda(x : String).\ x$ according to type translation.

By (GetLeft),

$$\vdash_{\text{get}} (\{name : String\} \,\&\, \{age : Int\}; name) : String$$

By typing rules, the translation of the program is

$$(\text{"}Isaac\text{"}, 10)$$

. If we apply the coercion to it, we get

$$\text{"}Isaac\text{"}$$

"put" Rules bruno: Missing example (and derivation)

The "put" judgment deals spefically with record update can be thought as producing a field updater. Compared to the "get" rules, the "put" rules take an extra input e, which is the desired expression to replace the field lablled l in values of type τ. (PutBase) is the base case. This rule allows refinement of record fields in the sense that the type of e can be a subtype of the type of the field labelled by l. The resulting type is $\{l : \tau'\}$ and the generated coercion is a constant function that always returns E. (PutLeft) and (PutRight) are complementary to each other: the idea is exactly the same as (GetLeft) and (GetRight) except that the refined type τ_1' and τ_2' is used.

5.5 Meta-theory

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* $\tau_1 <: \tau_3$.

Lemma 3. *If*

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

then

$$[\![\Gamma]\!] \vdash C : [\![\tau_1]\!] \to [\![\tau_2]\!]$$

Lemma 4 (Get rules produce the type-correct coercion.)**.** *If*

$$\Gamma \vdash_{get} \tau; l = C; \tau_1$$

then

$$[\![\Gamma]\!] \vdash C : [\![\tau]\!] \to [\![\tau_1]\!]$$

Proof. By induction on the given derivation. □

Lemma 5 (Put rules produce the type-correct coercion.)**.** *If*

$$\Gamma \vdash_{put} \tau; l; E = C; \tau_1$$

then

$$[\![\Gamma]\!] \vdash C : [\![\tau]\!] \to [\![\tau]\!]$$

Proof. By induction on the given derivation. □

Lemma 6 (Translation preserves well-formedness.)**.** *If*

$$\Gamma \vdash \tau$$

then

$$[\![\Gamma]\!] \vdash [\![\tau]\!]$$

Proof. By induction on the given derivation. □

Theorem 1 (Type preserving translation.)**.** *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. □

Type-Directed Translation to System F. Main results: type-preservation + coherence.

6. Implementation

We implemented all the functionalities of the $F_{\&}$ as a contribution to the open source community. Besides, we built a language on top of $F_{\&}$ to facilitate programming. We adopted a three-phase design to compile source programs built into System F terms.

1. A *type checking* phase that checks the usage of $F_{\&}$ and other source features against an abstract syntax tree that follows strictly with the source syntax. We intentionally made type checking happen first, so as to make the error messages relevant for programmers, although such an approach has obviously complicated the implementation.
2. A *desugaring* phase that translates well-typed source terms into $F_{\&}$ terms. A number of source-level features such as multi-field records, recursive `let` definitions, type synonyms are rewritten at this phase. The resulting program is just $F_{\&}$ expressions with other minor features.
3. A *compilation* phase that translates well-typed $F_{\&}$ terms into System F ones.

Phase 3 is what we have formalized in this paper.

Type synonyms. We implement *type synonym* as in Scala and Haskell.

In fact, to make future adaptations easier, we implemented this feature on top of the more general System F_ω.

```
type T[A, B] = t
e
```

6.1 Optimization.

The reader might argue that translating merges as pairs can be inefficient in practice. For example, the ternary merge

```
1,,2,,3
```

is elaborated into a nested tuple

```
((1,2),3)
```

as the merge operator associates to the left.

Another issue is that every time a coercion is generated, an application will be incurred at run-time.

7. Related work

Intersection types with polymorphism. Intersection types date back to as early as Coppo et al. [8]. Recently, some form of intersection types have been adopted in object-oriented languages such as Scala, Ceylon, and Grace. One defining difference, among others, is that all those languages only allow intersections of concrete types (classes), whereas our language allows intersections of type variables, such as `A & B`. Without that vehicle, we would not be able to define the generic `merge` function (below) for all interpretations of a given algebra, and would incur boilerplate code:

```
let merge [A, B] (f: ExpAlg A) (g: ExpAlg 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
}
```

In Scala community, there have been attempts to provide a foundational calculus for Scala that incorporates intersection types [1, 2].

Our type system combines intersection types and polymorphism. The closest to ours is Pierce's work [20] on a prototype compiler for a language with both intersection types, union types, and parametric polymorphism. The important difference with our system is that in his language 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. Pierce has also studied a system where both intersection types and bounded polymorphism are present in his Ph.D dissertation [21] and a 1997 report [22]. Going in the direction of higher kinds, Compagnoni and Pierce [7] 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. Compared to our work, they do not have a term-level construct for intersection introduction. Davies and Pfenning [11] 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.

Other type systems with intersection types. Dunfield [13] 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.

Reynolds invented Forsythe [23] in the 1980s. Our merge operator is analogous to his p_1, p_2. Castagna, and Dunfield describe elaborating multi-fields records into merge of single-field records. As Dunfield has noted, in Forsythe merges can be only used unambiguously. [5] For instance, it is not allowed in Forsythe to merge two functions.

Refinement intersection [10, 12, 15] 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. [16] study semantic subtyping.

Type systems for modularity. To address the problem of extensibility, some researchers have designed new type system features such as virtual classes [14], polymorphic variants [17], while others have shown employing programming pattern such as object algebras [19] by using features within existing programming languages. Both of the two approaches have drawbacks of some kind. The first approach often involves heavyweight designs, while the second approach still sacrifices the readability for extensibility.

Extensible records. Understanding records is important for understanding object-oriented languages. And we are the first to elaborate records to System F. Encoding records using intersection types appear in Reynolds [23] and Castagna et al. [5]. Although Dunfield also discusses this idea in his paper [13], he only provides an implementation but not formalization. Very similar to our treatment of elaborating records is Cardelli's work [3] 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 [9] 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 [24] started the work on extensible records and proposes row types [25] for records. Cardelli and Mitchell [4] define three primitive operations on records that are different from ours: *selection*, *restriction*, and *extension*. Following this approach, Leijen [18] define record update in terms of restriction and extension, while in our record system record update is a primitive operation. 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 en-

[5] Why the restriction?

coding an intersection type using first-class labels. Chlipala's `Ur` [6] explains record as type level constructs.

Indeed, our system can be adapted to simulate systems that support extensible records but not intersection of ordinary types like `Int` and `Float` by allowing only intersection of record types.

$\vdash_{\text{rec}} \tau$ states that τ is a record type, or the intersection of record types, and so forth.

$$\vdash_{\text{rec}} \{l : \tau\} \qquad \text{(RecBase)}$$

$$\frac{\vdash_{\text{rec}} \tau_1 \qquad \vdash_{\text{rec}} \tau_2}{\vdash_{\text{rec}} \tau_1 \,\&\, \tau_2} \qquad \text{(RecStep)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \hookrightarrow E_1 \qquad \vdash_{\text{rec}} \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \hookrightarrow E_2 \qquad \vdash_{\text{rec}} \tau_2}{\Gamma \vdash e_1,, e_2 : \tau_1 \,\&\, \tau_2 \hookrightarrow \langle E_1, E_2 \rangle} \qquad \text{(Merge')}$$

Of course our approach has its limitation as duplicated labels in a record are allowed. This has been discussed in a larger issue by Dunfield [13].

8. Conclusions and Further Work

Acknowledgments

Acknowledgments, if needed.

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

[17] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, volume 13. Baltimore, 1998.

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

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

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

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

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

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

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

[25] 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.