fi

Subtitle Text, if any

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

This is the text of the abstract.

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

General Terms Design, Languages, Theory

Keywords Intersecion Types, Polymorphism, Type System

1. Introduction

– Compare Scala: – merge[A,B] = new A with B
– type IEval = eval : Int – type IPrint = print : String
– 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:

- elaboration typing rules which given a source expression with intersection types, typecheck and translate it into an ordinary F term. Prove a type preservation result: if a term e has type τ in the source language, then the translated term $[\![e]\!]$ is well-typed and has type $[\![\tau]\!]$ in the target language.
- present an algorithm for detecting incoherence which can be very important in practice.
- explores the connection between intersection types and object algebra by showing various examples of encoding object algebra with intersection types.

2. A Taste of fi

[1]hange the examples later to something very simple.

This section provides the reader with the intuition of **fi** , while we postpone the presentation of the details in later sections.

In short, **fi** generalizes System F by adding intersection polymorphism. **fi** terms are elaborated into **f**, a variant of System F. System F, or polymorphic lambda calculus lays the foundation of functional programming languages such as Haskell.

The type system of **fi** permits a subtyping relation naturally and enables prototype-based inheritance. We will explore the usefulness of such a type system in practice by showing various examples.

2.1 Intersection Types

The central addition to the type system of **f** in **fi** is intersection types. What is an intersection type? One classic view is from set-theoretic interpretation of types: $A \wedge B$ stands for the intersection of the set of values of A and B. The other view, adopted in this paper, regards types as a kind of interface: a value of type $A \wedge B$ satisfies both of the interfaces of A and B. For example, `eval : Int` is the interface that supports evaluation to integers, while `eval : Int & print : String` supports both evaluation and pretty printing. Those interfaces are akin to interfaces in Java or traits in Scala. But one key difference is that they are unnamed in **fi** .

Intersection types provide a simple mechanism for ad-hoc polymorphism, similar to what type classes in Haskell achieve. The key constructs are the "merge" operator, denoted by ,,, at the value level and the corresponding type intersection operator, denoted by, $\wedge$ at the type level.

For example, we can define an (ad-hoc)-polymorphic `show` function that is able to convert integers and booleans to strings. In **fi** such function can be given the type

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

and be defined using the merge operator ,, as

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

where `showString` and `showBool` are ordinary monomorphic functions. Later suppose the integer `1` is applied to the `show` function, the first component `showInt` will be picked because the type of `showInt` is compatible with `1` while `showBool` is not.

2.2 Encoding Records

In addition to introduction of record literals using the usual notation, **fi** support two more operations on records: record elimination and

[1] C

```
def getName[A <: { def name: String }](v: A) = v.name
case class Person(name: String)
getName[Person](Person("Ek"))

let getName A (v : A & {name:String}) = v.name in
getName [{name:String}] {name = "Ek", age = 10}
```

Figure 1: Example of encoding records

record update.

A record type of the form $\{l : \tau\}$ can be thought as a normal type `t` tagged by the label `l`.

`e1` and `e2` are two expressions that support both evaluation and pretty printing and each has type `eval : Int, print : String`. `add` takes two expressions and computes their sum. Note that in order to compute a sum, `add` only requires that the two expressions support evaluation and hence the type of the parameter `eval : Int`. As a result, the type of `e1` and `e2` are not exactly the same with that of the parameters of `add`. However, under a structural type system, this program should typecheck anyway because the arguments being passed has more information than required. In other words, `eval : Int, print : String` is a subtype of `eval : Int`.

How is this subtyping relation derived? In **fi** , multi-field record types are excluded from the type system because `eval : Int , print : String` can be encoded as `eval : Int & print : String`. And by one of subtyping rules derives that `eval : Int & print : String` is a subtype of `eval : Int`.

2.3 Parametric Polymorphism

The presence of both parametric polymorphism and intersection is critical, as we shall see in the next section, in solving modularity problems. Here is a code snippet from the next section (The reader is not required to understand the purpose of this code at this stage; just recognizing the two types of polymorphism is enough.)

```
type SubExpAlg E = (ExpAlg E) \& { sub : E -> E -> E };
let e2 E (f : SubExpAlg E) = f.sub (exp1 E f) (f.lit 2);
```

`SubExpAlg` is a type synonym (a la Haskell) defined as the intersection of `ExpAlg E` and `sub : E ->E ->E`, parametrized by a type parameter `E`. `e2` exhibits parametric polymorphism as it takes a type argument `E`.

3. Application

```
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
}
```

3.1 Overloading

Dunfield [12] notes that using merges as a mechanism of overloading is not as powerful as type classes.

3.2 F-bounded Polymorphism

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, **fi** 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.

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)

3.3 Object Algebras

Object algebras provide an alternative to *algebraic data types* (ADT). 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 **fi** 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. [2]

The value level counterpart are functions of the type `A => B => A with B`. [3]

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 **fi** .

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 },
```

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

[3] FIXME

```
  add = \(x : IEval). \(y : IEval). { eval = x.eval + y.
      eval }
};
```

The interface is just a type synonym `IEval`. In **fi** , record types are structural and hence any value that satisfies this interface is of type `IEval` or of a subtype of `IEval`. [4]

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 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
```

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

3.4 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
}
```

[4] Should be mentioned in S2.

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 [5]. As `SubExp[A]` extends `Exp[A]`, the former becomes a subtype of the latter.

Such limitation does not exist in **fi** . 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.

3.5 Yanlin Stuff

This subsection presents yet another lightweight solution to the Expression Problem, inspired by the recent work by Wang. It has been shown that contravariant return types allows refinement of the types of extended expressions.

First, we define the type of expressions that support evaluation and implement two constructors:

```
type Exp = { eval: Int }
let lit (n: Int) = { eval = n }
let add (e1: Exp) (e2: Exp)
  = { eval = e1.eval + e2.eval }
```

If we would like to add a new operation, say pretty printing, it is nothing more than refining the original `Exp` interface by *intersecting* the original type with the new `print` interface using the `&` primitive and *merging* the original data constructors using the `,,` primitive.

```
type ExpExt = Exp & { print: String }
let litExt (n: Int) = lit n ,, { print = n.toString() }
let addExt (e1: ExpExt) (e2: ExpExt)
  = add e1 e2 ,,
    { print = e1.print.concat(" + ").concat(e2.print) }
```

Now we can construct expressions using the constructors defined above:

[5] It is still possible to encode contravariant parameter types in Scala but doing so would require some technique.

```
let e1: ExpExt = addExt (litExt 2) (litExt 3)
let e2: Exp = add (lit 2) (lit 4)
```

`e1` is an expression capable of both evaluation and printing, while `e2` supports evaluation only.

We can also add a new variant to our expression:

```
let sub (e1: Exp) (e2: Exp) = { eval = e1.eval - e2.eval
    }
let subExt (e1: ExpExt) (e2: ExpExt)
  = sub e1 e2 ,, { print = e1.print.concat(" - ").concat
      (e2.print) }
```

Finally we are able to manipulate our expressions with the power of both subtraction and pretty printing.

```
(subExt e1 e1).print
```

3.6 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.

fi 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
  = \ (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 fi calculus

[6]

This section formalizes the syntax, subtyping, and typing of **fi** . In the next section, we will go through the type-directed translation from **fi** to System F.

[6] Joshua Dunfield

Types

τ	$::=$	$\alpha \mid \tau \to \tau \mid \forall\alpha.\tau$	
	$\mid$	$\tau \wedge \tau$	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 introduction
	$\mid$	$e.l$	Record elimination
	$\mid$	e `with` $\{l = e\}$	Record update

Contexts

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

Labels

l

Figure 2: Syntax of **fi**

4.1 Syntax

Figure 2 specifies the syntax of **fi**. It extends the syntax of System F by adding the two features: intersection types and records. The formalization includes only single records because single record types as the multi-records can be desugared into the merge of multiple single records.

Types τ have five constructs. The first three are standard (present in System F): type variable α, function types $\tau \to \tau$, and type abstraction $\forall\alpha.\tau$; while the last two, intersection types $\tau \wedge \tau$ and record types $\{l : \tau\}$, are novel in **fi**. In record types, l is the label and τ the type.

First five constructs of expressions are also standard: variables x and two abstraction-elimination pairs. $\lambda(x\!:\!\tau).\, e$ abstracts expression e over values of type τ and is eliminated by application $e\ e$; $\Lambda\alpha.\, e$ abstracts expression e over types and is eliminated by type application $e\ \tau$.

The last four constructs are novel. $e,, e$ is the *merge* of two terms. $\{l = e\}$ introduces a record literal having l as the label for field containing expression e . $e.l$ access the field with label l in e. Finally, e `with` $\{l = e_1\}$ is a *new* record which is exactly the same as e except the field labelled l is updated to become $e1$. For simplicity, we omit other constructs in order to focus on the essence of the calculus. For example, fixpoints can be added in standard ways.

The field F is non-standard and introduced to deal with records. It is an associative list. Each item is a pair whose first item is either empty or a label and the second the types.

4.2 Subtyping

Subtyping is transitive and forms a lattice? Do we have a subtyping lattice?

Thanks to intersection types, we have natural subtyping relations among types. For example, $Int \wedge Bool$ should be a subtype of Int, since the former can be viewed as either Int or $Bool$. The subtyping rules are standard except for three points listed below:

1. $\tau_1 \wedge \tau_2$ is a subtype of τ_3, if *either* τ_1 or τ_2 are subtypes of τ_3,
2. τ_1 is a subtype of $\tau_2 \wedge \tau_3$, 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.

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

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

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

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

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

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

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

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

Figure 3: Subtyping

The first point is captured by two rules (S-And-1) and (S-And-2), whereas the second point by (S-And-3). Note that the last point means that record types are covariant in the type of the fields.

4.3 Typing

The typing judgment for **fi** is of the form: $\Gamma \vdash e : \tau$. This judgment uses the context Γ. The typing rules for our core languages are mostly standard ones for System F. In particular we introduce a (T-Merge) rule that applies to *merge* constructs.

dom reads: "the domain of". $F(l)$ means the result of lookup for l inside the associative list F. The order of lookup can be either from left to right or from right to left but has to be consistent inside one implementation. We prefer the order from the right to the left because it make possible record overriding. For example, $(\{count = 1\},,\{count = 2\}).count$ will evaluate to 2 in this case.

5. Type-directed Translation to System F

In this section we define the semantics of **fi** by means of a type-directed translation to System F. 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 **fi** 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 (RcdIntro). Hence $\{eval = 4\},,\{print = ``4''\}$ becomes $(4, ``4'')$.

Finally, $e1$ and $e2$ are both coerced by a projection function $(x : (Int, String)).\texttt{fst}\ x$.

5.2 Target Language

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

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

The dynamic semantics of System F can be found in ...

Lemma 1. *If*

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

then

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

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

We also define the type translation function $[\![\cdot]\!]$ from **fi** 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

$\boxed{\Gamma \vdash t}$

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

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

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

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

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

Figure 4: Well-formedness

coercion C is an expression in the target language and has type $\tau_1 \to \tau_2$, as proved by Lemma 1. 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 $\tau_1 \wedge \tau_2$ is also a subtype of τ_3 and the new coercion is a function that takes a value x of type $\tau_1 \wedge \tau_2$, project x on the first item, and apply C to it. (SAnd3) uses both of two coercions and constructs a pair.

5.4 Typing (Translation)

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

- **Coercion**

  Explained in the previous subsection.

- **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. (RcdIntro) uses the same System F expression E for e as for $\{l = e\}$. And in (RcdEim) and (RcdUpd) the coercions generated by the "get" and "put" rules will be used to coece the main **fi** expression.

  (RcdElim) 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 $C\ E$.

  (RcdUpd) is similar to (RcdElim) 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:

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

  The lambda is the field accessor and when applied to a translated expression of type $\{eval : Int\}$, 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

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

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

  By (GetLeft),

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

  By typing rules, the translation of the program is

  $$(``Isaac'', 10)$$

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash e_1,, e_2 : \tau_1 \wedge \tau_2} \quad \text{(Merge)}$$

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

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

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : \tau_1 \\ \vdash_{\text{put}} (t; l; e_1 : \tau_1) : (\tau_1', \tau') \qquad \tau_1 <: \tau_1'\end{array}}{\Gamma \vdash e\ \texttt{with}\ \{l = e_1\} : \tau'} \quad \text{(RcdUpd)}$$

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

The field with label l inside τ_1 is of type τ_2.

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

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

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

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

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

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

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

Figure 5: Typing

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

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

Figure 6: Type translation

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

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

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

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

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

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

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

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

Figure 7: Coercion

. If we apply the coercion to it, we get

$$``Isaac''$$

- **"put" rules**

  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 2. *Subtyping is reflexive Given a type τ, $\tau <: \tau$.*

Lemma 3. *Subtyping is transitive If $\tau_1 <: \tau_2$ and $\tau_2 <: \tau_3$, then $\tau_1 <: \tau_3$.*

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*

$$\Gamma \vdash e : \tau \hookrightarrow E$$

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \hookrightarrow E_1 \\ \Gamma \vdash e_2 : \tau_2 \hookrightarrow E_2\end{array}}{\Gamma \vdash e_1,, e_2 : \tau_1 \wedge \tau_2 \hookrightarrow \langle E_1, E_2 \rangle} \quad \text{(Merge)}$$

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

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

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \hookrightarrow E \qquad \Gamma \vdash e_1 : \tau_1 \hookrightarrow E_1 \\ \vdash_{\text{put}} (t; l; e_1 : \tau_1 \hookrightarrow E_1) : (\tau_1', \tau') \hookrightarrow C \qquad \tau_1 <: \tau_1'\end{array}}{\Gamma \vdash e\ \texttt{with}\ \{l = e_1\} : \tau' \hookrightarrow C\ E} \quad \text{(RcdUpd)}$$

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

The field with label l inside τ_1 is of type τ_2.

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

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

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

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

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

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

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

Figure 8: Type-directed translation from **fi** to System F.

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

Our implementation currently supports multi-field records.

6.1 Type Synonyms

We extend the implementation of the type system extended with type synonyms and lazy arguments.

```
type T A1 A2 = ... in
```

6.2 Optimization

7. Related Work

Intersection types date back to Coppo et al. [7]. From another viewpoint, one can regard intersections as "implicit pairs" whose introduction is explicit by using the merge operator and elimination is implicit (with no source-level construct for elimination). Understanding records is important for understanding object-oriented languages. And we are the first to elaborate records in terms of System F.

[3]

Intersection Types with Polymorphism. Our type system combines intersection types and polymorphism. The closest to ours is Pierce's Ph.D. thesis [17] 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. However, as shown in Section 3, this feature is critical in supporting modularity and extensibility because it allows dynamic composition of values.

Other Type Systems with Intersection Types. Dunfield [12] 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. Although similar in spirit, our elaboration typing is simpler: we require subtyping in the case of applications, thus avoiding the subsumption rule. Besides, our treatment combines the merge rules (k ranges over $\{1, 2\}$)

$$\frac{\Gamma \vdash e_k : \tau}{\Gamma \vdash e_1,, e_2 : \tau}$$

and the standard intersection-introduction rule

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash e : \tau_2}{\Gamma \vdash e : \tau_1 \wedge \tau_2}$$

into one rule:

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

Reynolds invented Forsythe in the 1980s. Our merge operator is analogous to 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. For instance, it is not allowed in Forsythe to merge two functions. Reynolds and Castagna do not consider elaboration and Dunfield do not formalize elaborating records.

Both Pierce and Dunfield's system include a subsumption rule, which states that if an expression has been inferred of type τ, then it is also of any supertype of τ. Our system does not have this rule.

What's the implication of refinement intersection being type-checkers?

Refinement intersection [11, 13] 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", as Dunfield [12] has noted, it is not possible to encode various language features. [9] [11]

Semantic subtyping. Frisch et al. [14]

Type Systems for Modularity. Intersection types have been shown to be useful in designing languages that support modularity. [15]

[16]

Encoding Records and Objects. Extensible records were introduced by Wand [20]. Encoding records using intersection types appear in Reynolds [19] and Castagna et al. [5]. Although Dunfield also discusses this idea in his paper [12] and provides an implementation, he does not proceed to propose a set of elaboration typing rules for records. Very similar to our treatment of elaborating records is Cardelli's work [4] 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 [8] 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.

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.

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

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

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

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 [12].

Rémy [18]

Other Related Work. Compagnoni and Pierce [6] add intersection types to System F^ω and use the new calculus, $F^\omega_\wedge$, to model multiple inheritance. Compared to F^ω, in their system, types additionally 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 [10] 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. In the Scala community, there have been attempts to provide a foundational calculus for Scala that incorporates intersection types [1, 2].

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] F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

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

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

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

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

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

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

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

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

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

[14] 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.

[15] 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.

[16] B. C. d. S. Oliveira, T. Van Der Storm, A. Loh, and W. R. Cook. Feature-oriented programming with object algebras. In *ECOOP 2013–Object-Oriented Programming*, pages 27–51. Springer, 2013.

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

[18] D. Rémy. Type checking records and variants in a natural extension of ml. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–88. ACM, 1989.

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

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