fi

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

Keywords intersecion types, inheritance

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 **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 \& 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 \& 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, "`\&`" 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 record update.

A record type of the form `l : t` 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`.

[1] C

`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

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)

3.1 Object Algebras

Object algebra provides 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`.

Our type system is an fairely 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 **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 },
  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`.

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
```

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(
  sub = \(x : IPrint). \(y : IPrint). { print = x.print.concat(
};
```

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

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

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.

3.2 Visitors

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 [3]. 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:

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)
};
```

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

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.

3.3 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:

```
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.pri
```

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

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

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.

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. Source Language

The semantics of the source-level language is not defined formally, instead by a translation into the target language.

The most central construct of our language is ...

The source language, System FI, is identical to the source language described in the previous section, except for the two additions: intersection types and records. The formalization includes only single records and single record types as the multi-records can be desugared into the merge of multiple single records.

Dunfield has described a language that includes a "top" type but it does not appear in our language. Our work differs from Dunfield in that ...

Remark. The operational semantics of FI is not presented in this paper. However,

4.1 Source Syntax

4.2 Source Subtyping

4.3 Source Typing

5. Elaboration Typing

5.1 Target Language

The target language is System F extended with a base type Int. The syntax and typing is completely standard. The types are function, universal quantification.

In order to give the reader an intuitive idea of how the elaboration works, let's first imagine a manual translation.

First, multi-field record literals are desugared into merges of single-field record literals. Therefore $\{eval = 4, print = \text{"}4''\}$ becomes $\{eval = 4\},, \{print = \text{"}4''\}$. Merges of two values are elaborated into just a pair of them and single-field record literals lose their field labels during the elaboration. Hence $\{eval = 4\},, \{print = \text{"}4''\}$ becomes $(4, \text{"}4'')$.

Finally, $e1$ and $e2$ are both coerced by a projection function $(x : (Int, String)).x._1$ before being applied to add. We adopt a Scala-like syntax where $._1$ denotes the projection of a tuple on the first element, and so on.

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

$$
\begin{array}{rcl}
|\alpha| & = & \alpha \\
|\tau_1 \to \tau_2| & = & |\tau_1| \to |\tau_2| \\
|\forall \alpha.\tau| & = & \forall \alpha.|\tau| \\
|t_1 \& t_2| & = & \langle |\tau_1|, |\tau_2| \rangle \\
|\{l : \tau\}| & = & |\tau|
\end{array}
$$

Lemma 1. *If*

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

then

$$|\Gamma| \vdash C : |\tau_1| \to |\tau_2|$$

In this section, we present a relatively lightweight type-directed elaboration from FI to F. The elaboration consists of four sets of rules, which are explained below:

- **Coercion**

  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, which is just an expression in the target language, is guaranteed to have 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 t_1 that is a subtype of the type of t_2, the elaborated e, when applied to the corresponding coercion C, has exactly type $|t_2|$". 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 `TrApp` case.

- **Elaboration**

  The elaboration judgment $\Gamma \vdash e : \tau \hookrightarrow E$ extends the typing judgment with an elaborated expression on the right hand side of $\hookrightarrow$. It is also standard, except for the case of `TrApp`, in which a coercion from the inferred type of the argument to the expected type of the parameter is inserted before the argument; and the case of `TrRcdEim` and `TrRcdUpd`, where the "get" and "put" rules will be used. The two set of rules are explained below.

- **"get" rules**

  The "get" judgment can be thought as producing a field accessor.

- **"put" rules**

  The "put" judgment can be thought as producing a field updater.

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

6. Implementation

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

- **Elaborating simply-typed lambda calculus**

  Dunfield has introduced a type system with intersection polymorphism but no parametric polymorphism.

Nystrom et. al. OOPSLA 06

Applications:

- Object/Fold Algebras. How to support extensibility in an easier way.

See Datatypes a la Carte

- Mixins

- Lenses? Can intersection types help with lenses? Perhaps making the types more natural and easy to understand/use?

- Embedded DSLs? Extensibility in DSLs? Composing multiple DSL interpretations?

http://www.cs.ox.ac.uk/jeremy.gibbons/publications/embedding.pdf

8. Conclusion

Acknowledgments, if needed.

A. Proofs

Proof. By structural induction on the types and the corresponding inference rule.

`(SubVar)`
`(SubFun)`
`(SubForall)`
`(SubAnd1)`
`(SubAnd2)`
`(SubAnd3)`
`(SubRcd)`

□

Lemma 2. *If*

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

then

$$|\Gamma| \vdash C : |\tau| \to |\tau_1|$$

Proof. By structural induction on the type and the corresponding inference rule.

`(Get-Base)` $\Gamma \vdash_{get} \{l : \tau\}; l = \lambda(x : |\{l : \tau\}|).x; \tau$

By the induction hypothesis

$$|\Gamma| \vdash \lambda(x : |\{l : \tau\}|).x : |\{l : \tau\}| \to |\tau|$$

`(Get-Left)`
`(Get-Right)`

□

Lemma 3. *If*

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

then

$$|\Gamma| \vdash C : |\tau| \to |\tau|$$

Proof. By structural induction on the type and the corresponding inference rule.

`(Put-Base)`
`(Put-Left)`
`(Put-Right)`

□

Lemma 4. *If*

$$\Gamma \vdash \tau$$

then

$$|\Gamma| \vdash |\tau|$$

Proof. Since

$$\Gamma \vdash \tau$$

It follows from `(FI-WF)` that

$$\mathrm{ftv}(\tau) \subseteq \mathrm{ftv}(\Gamma)$$

And hence

$$\mathrm{ftv}(|\tau|) \subseteq \mathrm{ftv}(|\Gamma|)$$

By `(F-WF)` we have

$$\Gamma \vdash \tau$$

□

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

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

then

$$|\Gamma| \vdash E : |\tau|$$

Proof. By structural induction on the expression and the corresponding inference rule.

`(TrVar)` $\Gamma \vdash x : \tau \hookrightarrow x$

It follows from `(TrVar)` that

$$(x : \tau) \in \Gamma$$

Based on the definition of $|\cdot|$,

$$(x : |\tau|) \in |\Gamma|$$

Thus we have by `(F-Var)` that

$$|\Gamma| \vdash x : |\tau|$$

`(TrAbs)` $\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2 \hookrightarrow \lambda x : |\tau_1|.E$

It follows from `(TrAbs)` that

$$\Gamma, x : \tau_1 \vdash e : \tau_2 \hookrightarrow E$$

And by the induction hypothesis that

$$|\Gamma|, x : |\tau_1| \vdash E : |\tau_2|$$

By `(TrAbs)` we also have

$$\Gamma \vdash \tau_1$$

It follows from Lemma 4 that

$$|\Gamma| \vdash |\tau_1|$$

Hence by `(F-Abs)` and the definition of $|\cdot|$ we have

$$|\Gamma| \vdash \lambda x : |\tau_1|.E : |\tau_1 \to \tau_2|$$

`(TrApp)` $\Gamma \vdash e_1 e_2 : \tau_2 \hookrightarrow E_1(CE_2)$

From `(TrApp)` we have

$$\Gamma \vdash \tau_3 <: \tau_1 \hookrightarrow C$$

Applying Lemma 1 to the above we have

$$|\Gamma| \vdash C : |\tau_3| \to |\tau_1|$$

Also from (TrApp) and the induction hypothesis

$$|\Gamma| \vdash E_1 : |\tau_1| \to |\tau_2|$$

Also from (TrApp) and the induction hypothesis

$$|\Gamma| \vdash E_2 : |\tau_3|$$

Assembling those parts using (F-App) we come to

$$|\Gamma| \vdash E_1(CE_2) : |\tau_2|$$

□

(TrTAbs) $\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau \hookrightarrow \forall\alpha.E$

From (TrTAbs) we have

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

By the induction hypothesis we have

$$|\Gamma| \vdash E : |\tau|$$

Thus by (F-TAbs) and the definition of $|\cdot|$

$$\Gamma \vdash \Lambda\alpha.E : |\forall\alpha.\tau|$$

(TrTApp) $\Gamma \vdash e\ \tau : [\alpha := \tau]\tau_1 \hookrightarrow E\ |\tau|$

From (TrTApp) we have

$$\Gamma \vdash e : \forall\alpha.\tau_1 \hookrightarrow E$$

And by the induction hypothesis that

$$|\Gamma| \vdash E : \forall\alpha.|\tau_1|$$

Also from (TrTApp) and Lemma 4 we have

$$|\Gamma| \vdash |\tau|$$

Then by (F-TApp) that

$$|\Gamma| \vdash E\ |\tau| : [\alpha := |\tau|]|\tau_1|$$

Therefore

$$|\Gamma| \vdash E\ |\tau| : |[\alpha := \tau]|\tau_1||$$

From (TrMerge) and the induction hypothesis we have

$$|\Gamma| \vdash E_1 : |\tau_1|$$

and

$$|\Gamma| \vdash E_2 : |\tau_2|$$

Hence by (F-Pair)

$$|\Gamma| \vdash \langle E_1, E_2\rangle : \langle|\tau_1|, |\tau_2|\rangle$$

Hence by the definition of $|\cdot|$

$$|\Gamma| \vdash \langle E_1, E_2\rangle : |\tau_1 \& \tau_2|$$

(TrRcdIntro) $\Gamma \vdash \{l = e\} : \{l : \tau\} \hookrightarrow E$

From (TrRcdIntro) we have

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

And by the induction hypothesis that

$$|\Gamma| \vdash E : |\tau|$$

Thus by the definition of $|\cdot|$

$$|\Gamma| \vdash E : |\{l : \tau\}|$$

(TrRcdElim) $\Gamma \vdash e.l : \tau_1 \hookrightarrow CE$

From (TrRcdElim)

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

And by the induction hypothesis that

$$|\Gamma| \vdash E : |\tau|$$

Also from (TrRcdEim)

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

Applying Lemma 2 to the above we have

$$|\Gamma| \vdash C : |\tau| \to |\tau_1|$$

Hence by (F-App) we have

$$|\Gamma| \vdash CE : |\tau_1|$$

From (TrRcdUpd)

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

And by the induction hypothesis that

$$|\Gamma| \vdash E : |\tau|$$

Also from (TrRcdUpd)

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

Applying Lemma 3 to the above we have

$$|\Gamma| \vdash C : |\tau| \to |\tau|$$

Hence by (F-App) we have

$$|\Gamma| \vdash CE : |\tau|$$

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...
Coppo, M., Dezani-Ciancaglini, M.: A new type-assignment for -terms. Archiv. Math. Logik 19, 139156 (1978)