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 improve on those designs. In particular, System FI has two main advantages over existing languages:

- It supports dynamic composition of intersecting values.
- 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 in statically typed functional languages.

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

we first define an interface that supports the evaluation operation:

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

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.

```
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 }
};
```

`evalAlg` has type `ExpAlg Int`

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

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

```
let e1 E (f : ExpAlg E) = f.add (f.lit 6) (f.lit 6);
let e2 E (f : SubExpAlg E) = f.sub (e1 E f) (f.lit 2);
```

defines two expressions. There are two interesting points that are worth noting here. First, the expressions themselves are functions that take an extra argument, the algebras, and use the data constructors in the object algebra (factory) `f` such as `lit`, `add` and `sub` to create values.

```
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
};

let newAlg = merge IEval IPrint subEvalAlg printAlg in

(e1 (IEval & IPrint) newAlg).print
```

`e1` is a single object created that supports both evaluation and printing, thus achieving full feature-oriented programming.

The merge operator , , is used in the definition of $merge$.

3.2 From Algebras Back to Visitors

The visitor pattern allows adding new operations to existing structures without modifying those structures.

```
type ExpAlg E = { lit : Int -> E, add : E -> E -> E };
type Exp = { accept : forall A. ExpAlg A -> A };

type LitAdd A = { lit : Int -> A, add : A -> A -> A };

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

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

let evalAlg = { lit = \(x : Int). x, add = \(x : Int).
    \(y : Int). x + y };
let e1 = add (lit 2) (lit 3);
-- e1.accept Int evalAlg

type SubExpAlg E = (ExpAlg E) & { sub : E -> E -> E };
type ExpExt = { accept : forall A. SubExpAlg A -> A };

type LitAddSub A = LitAdd A & { sub : A -> A -> A };

let sub (e1 : ExpExt) (e2 : ExpExt)
  = { accept = /\A. \(f : LitAddSub A). f.sub (e1.accept
      A f) (e2.accept A f) };

-- Contravariant param type, programmer-friendly usage
let e2 = sub (lit 2) (lit 3);
```

```
-- Note that Exp <: ExpExt
let f (x : ExpExt) = 1;
-- let g (x : Exp) = 1;
e2
```

Contravariant param type helpers programmers to write more intuitive programs.

3.3 Yanlin

3.4 Mixins

```
type Open S = 'S -> S;
-- Two ways of defining `fix` in Haskell:
--   let fix f = let x = f x in x
--   let fix f = f (fix f)
let rec fix S (f : Open S) : S = f (fix S f);
let fact (this : '(Int -> Int)) (n : Int) = if n == 0
    then 1 else n * this (n - 1);
type Exp = { print : 'String, eval : Int };
let e = { print = "1", eval = 1 };
let add (e1 : Exp) (e2 : Exp) : Open Exp = \(this : 'Exp
    ). { print = this.eval.toString(), eval = 1 };
(fix Exp (add e e)).print
```

In Haskell, one is able to write programs in mixin style using records. However, this approach has a serious drawback: 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.

3.5 Composing Mixins and Object Algebras

4. 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.

4.1 Target Syntax

4.2 Target Typing

5. Source Language

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,

5.1 Source Syntax

5.2 Source Subtyping

5.3 Source Typing

6. Elaboration Typing

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.

7. Implementation

7.1 Type Synonyms

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

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

7.2 Optimization

8. 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

9. Conclusion

Acknowledgments

Acknowledgments, if needed.

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)