FI

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

Keywords intersecion types, inheritance

1. Introduction

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

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

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.

This paper introduces System F_{IR}

2. Overview

There should be a section informally describing the language (System FI) through various examples. Intersection types provide a general mechanism for ad-hoc polymorphism.

While elaboration intersection types has been

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.

```
let max (x : Int) (y : Int) = if (x > y) then x else y
```

Mention properties informally via examples:

Typeclasses

Intersecion types in Scala

Examples in this paper are written with a Haskell-like syntax. Consider a `show` function that converts either integers or booleans to strings. In FI it can be given the type:

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

And can be defined as:

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

where `showString` and `showBool` are ordinary monomorphic functions.

The merge construct in the original function is elaborated into a pair in the target language:

```
show = (showInt, showBool)
```

In the target language where there is no intersection types, the application of the integer `1` to this function does not typecheck. However, we may rescue this situation by inserting a coercion that extracts the first item out of this pair.

Thus `show 1` in FI corresponds to `(fst show) 1` in F.

2.1 Algebras

2.2 Lenses

2.3 Embedded DSLs

3. System F

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. System FI

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.

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

5. Type-Directed Translation to System F

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 $\tau_1 \rightarrow \tau_2$, 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

We extend the implementation of the type system extended with type synonyms.

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

7. Case Study?

8. Properties

9. Case Studies

10. 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

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

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

Lemma 1. *If*

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

then

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

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

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

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

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

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

□

Lemma 3. *If*

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

then

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

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

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

(TrMerge) $\Gamma \vdash e_1,, e_2 : \tau_1 \& \tau_2 \hookrightarrow \langle E1, E2\rangle$

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

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

Hence by `(F-App)` we have

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

`(TrRcdUpd)` $\Gamma \vdash e$ with $\{l = e_1\} : \tau \hookrightarrow CE$

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

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

Hence by `(F-App)` we have

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

A. Proofs

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...