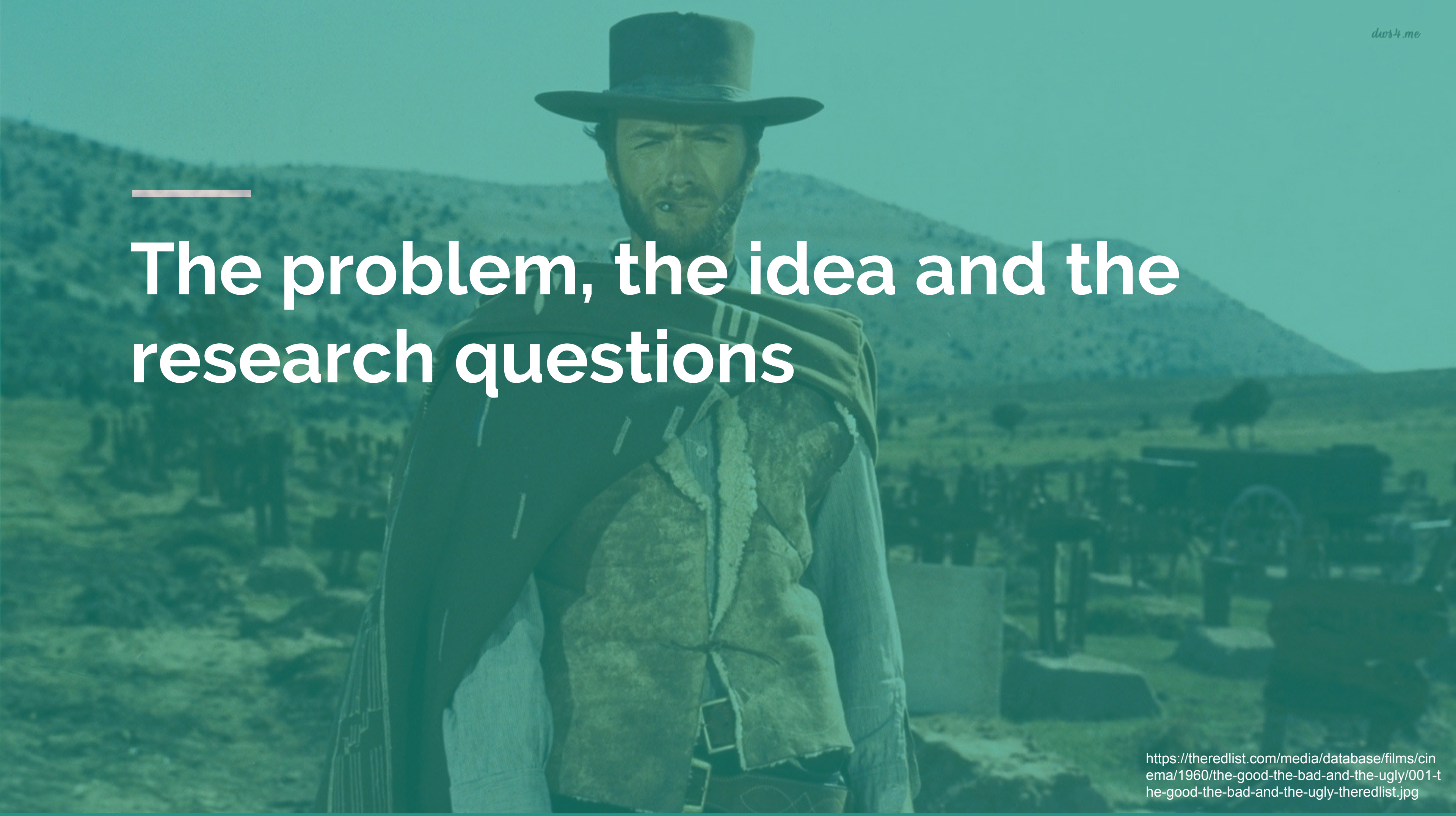
Algebraic Subtyping for Algebraic Effects and Handlers

Axel Faes
KULeuven

The problem, the idea and the research questions



https://theredlist.com/media/database/films/cinema/1960/the-good-the-bad-and-the-ugly/001-the-good-the-bad-and-the-ugly-theredlist.jpg

Problem

Algebraic effects and handlers

Formally model side-effects
(Matija Pretnar, Gordon Plotkin)

Existing type-&-effect systems
Awkward to implement
Theoretically unsatisfactory

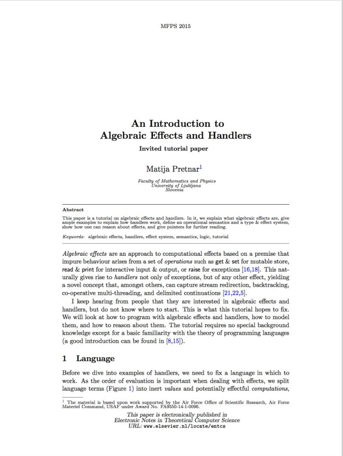
```
effect Decide : unit -> bool;;

let choose_all = handler
    | #Decide () k -> k true @ k false
    | val x -> [x];;

with choose_all handle (if #Decide () then 10 else 20)
(* Output: [10; 20] *)
```

Idea

Stephen Dolan

Subtyping + Parametric Polymorphism

Extend Dolan’s type system with effect information



MFPS 2015

An Introduction to
Algebraic Effects and Handlers

Invited tutorial paper

Matija Pretnar[1]

Faculty of Mathematics and Physics
University of Ljubljana
Slovenia

Abstract

This paper is a tutorial on algebraic effects and handlers. In it, we explain what algebraic effects are, give ample examples to explain how handlers work, define an operational semantics and a type & effect system, show how one can reason about effects, and give pointers for further reading.

Keywords: algebraic effects, handlers, effect system, semantics, logic, tutorial

Algebraic effects are an approach to computational effects based on a premise that impure behaviour arises from a set of *operations* such as get & set for mutable store, read & print for interactive input & output, or raise for exceptions [16,18]. This naturally gives rise to *handlers* not only of exceptions, but of any other effect, yielding a novel concept that, amongst others, can capture stream redirection, backtracking, co-operative multi-threading, and delimited continuations [21,22,5].

I keep hearing from people that they are interested in algebraic effects and handlers, but do not know where to start. This is what this tutorial hopes to fix. We will look at how to program with algebraic effects and handlers, how to model them, and how to reason about them. The tutorial requires no special background knowledge except for a basic familiarity with the theory of programming languages (a good introduction can be found in [8,15]).

1 Language

Before we dive into examples of handlers, we need to fix a language in which to work. As the order of evaluation is important when dealing with effects, we split language terms (Figure 1) into inert *values* and potentially effectful *computations*,

[1] The material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Polymorphism, Subtyping, and Type Inference in MLsub

Stephen Dolan Alan Mycroft
University of Cambridge, UK
stephen.dolan@cl.cam.ac.uk alan.mycroft@cl.cam.ac.uk

Abstract

We present a type system combining subtyping and ML-style parametric polymorphism. Unlike previous work, our system supports type inference and has compact principal types. We demonstrate this system in the minimal language MLsub, which types a strict superset of core ML programs.

This is made possible by keeping a strict separation between the types used to describe inputs and those used to describe outputs, and extending the classical unification algorithm to handle subtyping constraints between these input and output types. Principal types are kept compact by type simplification, which exploits deep connections between subtyping and the algebra of regular languages. An implementation is available online.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

Keywords Subtyping, Polymorphism, Type Inference, Algebra

1. Introduction

The Hindley-Milner type system of ML and its descendants is popular and practical, sporting decidable type inference and principal types. However, extending the system to handle subtyping while preserving these properties has been problematic.

Subtyping is useful to encode extensible records, polymorphic variants, and object-oriented programs, but also allows us to expose more polymorphism even in core ML programs that do not use such features. For instance, given the definition twice $= \lambda f.\lambda x.f(fx)$, ML gives twice $(\lambda x.\text{true})$ the principal type bool $\to$ bool, allowing only arguments of type bool, while MLsub gives it type $\top \to$ bool, allowing arguments of any type (see Section 2.4 for details).

Supporting subtyping requires thinking carefully about data flow. Consider the `select` function, which takes three arguments: a predicate p, a value v and a default d, and returns the value if the predicate holds of it, and the default otherwise:

$$\texttt{select}\ p\ v\ d = \texttt{if}\ (p\ v)\ \texttt{then}\ v\ \texttt{else}\ d$$

In ML and related languages, `select` has type scheme

$$\forall\alpha.\,(\alpha \to \texttt{bool}) \to \alpha \to \alpha \to \alpha$$

[Copyright notice will appear here once 'preprint' option is removed.]

This scheme is quite strange, in that it demands that whatever we pass as the default d be acceptable to the predicate p. But this constraint does not arise from the behaviour of the program: at no point does our function pass d to p.

Let’s examine the actual data flow of this function:

v d
argument to p result

Only by ignoring the orientation of the edges above could we conclude that d flows to the argument of p. Indeed, this is exactly what ML does: by turning data flow into equality constraints between types, information about the *direction* of data flow is ignored. Since equality is symmetric, data flow is treated as undirected.

To support subtyping is to care about the direction of data flow. With subtyping, a source of data must provide at least the guarantees that the destination requires, but is free to provide more.

The data flow graph of a program gives rise to constraints between types, which can be represented as a *constraint graph*. Once we add subtyping, we must treat the constraint graph as directed rather than undirected. This directed constraint graph complicates matters: instead of using unification to find connected components in the constraint graph, we need to deal with reachability and cycles and strong connectivity.

One way to sidestep these difficulties is to widen the definition of “type” to include an explicit constraint graph. This gives the notion of a *constrained type* used in previous work [7, 22], where the principal type of an expression is given as a type with an attached explicit graph of constraints. For instance, such a system might represent the type of `select` as follows:

$$(\alpha \to \texttt{bool}) \to \alpha \to \beta \to \gamma \mid \alpha \leq \gamma, \beta \leq \gamma$$

Unfortunately, this just moves the burden of dealing with constraint graphs onto the programmer. Constrained types are often large, and are difficult to compare.

In his PhD thesis, Pottier[1] noticed that constraint graphs can be simplified due to the structure of data flow. By keeping a strict separation between inputs and outputs, we can always represent the constraint graph as a *bipartite* graph: data flows from inputs to outputs. With edges only from inputs to outputs, such graphs have no cycles (or even paths of more than one edge), simplifying analysis.

We take this insight a step further, and show that by keeping the same religious distinction between input and output we can avoid using explicit constraint graphs altogether, giving a much simpler type system. In MLsub, `select` has this type scheme:

$$\forall\alpha,\beta.\,(\alpha \to \texttt{bool}) \to \alpha \to \beta \to (\alpha \sqcup \beta)$$

[1] The *mono-polarity invariant* [18, ch. 12]

1 2016/11/7

Research questions

How can Dolan’s elegant type system be extended with effect information?

Which properties are preserved and which aren’t preserved?

What advantages are there to an type-&-effect system based on Dolan’s elegant type system?

Planning

DONE

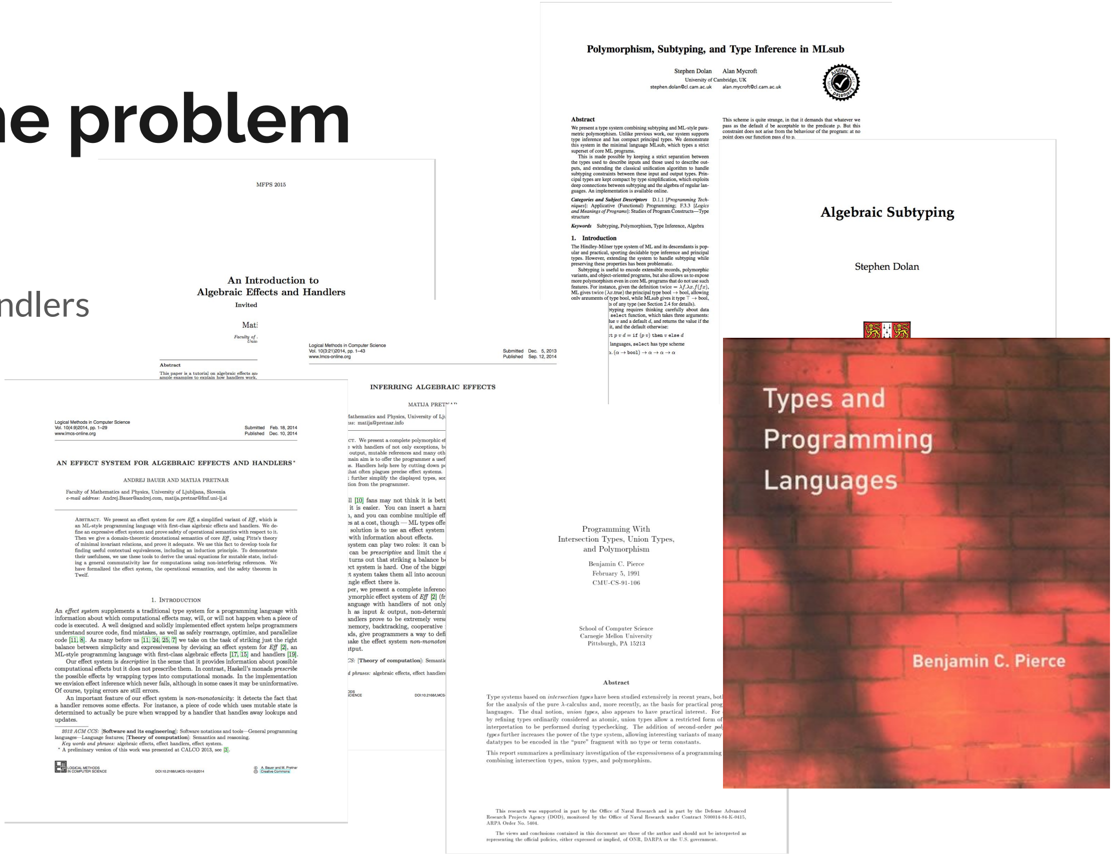
Understand the problem

Literature study

- Algebraic Effects and Handlers
- Dolan’s system

“Play” with implementations

Study intersection/union types



DONE BY END OCTOBER

Develop type system

Terms & Types

Subtyping rules

Typing rules

Semantics

Type inference algorithm

Constraint generation

(pure) type A, B	$::=$	$\mathsf{bool} \mid \mathsf{int}$	basic types
	$\mid$	$A \to \underline{C}$	function type
	$\mid$	$\underline{C} \Rightarrow \underline{D}$	handler type
	$\mid$	α	type variable
	$\mid$	$\forall \alpha . A$	polytype
	$\mid$	$\top$	top
	$\mid$	$\bot$	bottom
	$\mid$	$A \sqcap B$	intersection
	$\mid$	$A \sqcup B$	union
dirty type $\underline{C}, \underline{D}$	$::=$	$A \,!\, \Delta$	
	$\mid$	$\underline{C} \sqcap \underline{D}$	intersection
	$\mid$	$\underline{C} \sqcup \underline{D}$	union
dirt Δ	$::=$	$\{\mathrm{R}\}$	
R	$::=$	$\mathsf{Op} \,;\, R$	row
	$\mid$	δ	row variable
	$\mid$	$.$	closed row
	$\mid$	$R_1 \sqcap R_2$	intersection
	$\mid$	$R_1 \sqcup R_2$	union
All operations Ω	$::=$	$\{\mathsf{Op}_i \mid \mathsf{Op}_i \in \Sigma\}$	

typing contexts $\Gamma \ ::= \ \epsilon \mid \Gamma, x : A, x : \forall \alpha . B$

Expressions

Val
$$\frac{\Gamma \vdash v : A \qquad A \leqslant B}{\Gamma \vdash v : B}$$

Var
$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

Const
$$\frac{(\mathsf{k} : A) \in \Sigma}{\Gamma \vdash \mathsf{k} : A}$$

Type Abs
$$\frac{\Gamma, \alpha \vdash v : A}{\Gamma \vdash \Lambda \alpha . v : \forall \alpha . A}$$

Type App
$$\frac{\Gamma \vdash v : \forall \alpha . B}{\Gamma \vdash v\, A : B[A/\alpha]}$$

Fun
$$\frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \mathsf{fun}\ x : A \mapsto c : A \to \underline{C}}$$

Hand
$$\frac{\Gamma, x : A \vdash c_r : B \,!\, \Delta \qquad \left[\begin{array}{l} (\mathsf{Op} : A_{\mathsf{Op}} \to B_{\mathsf{Op}}) \in \Sigma \\ \Gamma, x : A_{\mathsf{Op}}, k : B_{\mathsf{Op}} \to B \,!\, \Delta \vdash c_{\mathsf{Op}} : B \,!\, \Delta \end{array} \right]_{\mathsf{Op} \in O}}{\Gamma \vdash \{\mathsf{return}\ x \mapsto c_r, [\mathsf{Op}\ y\ k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\} : A \,!\, \Delta \cup O \Rightarrow B \,!\, \Delta}$$

DEADLINE: MARCH

Theory

Proofs

- Instantiation / Weakening
- Substitution / Soundness
- Type preservation

Using Coq Proof Assistant

```
Inductive type_of: type_env -> expr -> type -> Prop :=
  type_of_const: ∀ env: type_env, ∀ n: nat, (type_of env (Const n) Nat)
| type_of_var: ∀ env: type_env, ∀ x: ident,
  ∀ t: type, ∀ ts: type_scheme,
  (assoc_ident_in_env x env)=(Some ts)  ->
    (is_gen_instance t ts) -> (type_of env (Variable x) t)
| type_of_lam: ∀ env: type_env, ∀ x: ident, ∀ e: expr, ∀ t, t': type,
  (type_of (add_env env x (type_to_type_scheme t)) e t') ->
    (type_of env (Lam x e) (Arrow t t'))
...
```

Proving ML Type Soundness Within Coq

Implementation

Implement in Eff

Write type inference engine

```
and type_expr st {Untyped.term=expr; Untyped.location=loc} = type_plain_e

(* Type a plain expression *)
and type_plain_expr st loc = function
  | Untyped.Var x ->
    let ty_sch, st = get_var_scheme_env ~loc st x in
    Ctor.var ~loc x ty_sch, st
  | Untyped.Const const ->
    Ctor.const ~loc const, st
  | Untyped.Tuple es ->
    let els = List.map (fun (e, _) -> e) (List.map (type_expr st) es) in
    Ctor.tuple ~loc els, st
  | Untyped.Record lst ->
    let lst = List.map (fun (f, (e, _)) -> (f, e)) (Common.assoc_map (typ
    Ctor.record ~loc lst, st
  | Untyped.Variant (lbl, e) ->
    let exp = Common.option_map (fun (e, _) -> e) (Common.option_map (typ
    Ctor.variant ~loc (lbl, exp), st
  | Untyped.Lambda (p, c) ->
    let pat = type_pattern st p in
    let comp, st = type_comp st c in
    Ctor.lambda ~loc pat comp, st
  | Untyped.Effect eff ->
    let eff = infer_effect ~loc st eff in
```

DEADLINE: MAY

Validation

Testing against other systems

- Coercions
- Subtyping
- Row polymorphism

Usecase

- Optimizations

http://cdn2.hubspot.net/hub/53/file-311443092-png/Blog-Related_Images/website-optimization.png

JUNE

Finish