Information Effects for Understanding Type Systems

Or: how someone else found the maths to justify my dogma

Philippa Cowderoy

August 23, 2016

What are effects?

Effects can be seen in relation to models of computing:

General computing	Anything to do with the real world
Functional programming	Mutation, control effects...
Total programming	Non-termination
Logic programming	Exposing the solver (eg cut)

Information Effects

Work on reversible programming at Indiana produced Π, a combinator language in which computing is performed by isomorphism. New effect class:

General computing	Anything to do with the real world
Functional programming	Mutation, control effects...
Total programming	Non-termination
Logic programming	Exposing the solver (eg cut)
Isomorphic programming	Information Effects

Named for conservation of information in quantum mechanics, information effects create or destroy information and thus violate isomorphism.

CLP in terms of Prolog

Constraint Logic Programming in a Prolog-like language can be achieved through two simple steps:

- Where Prolog would unify, introduce an equality constraint
- Describe new kinds of constraints and their solver, start introducing them

- Strictly optional: Interaction with the solver other than telling it about constraints or asking for the solution to a complete problem
- "Complete problem" might be a difficult (region-like!) notion if you're getting everything done through equality constraints per se though

Modes

- A predicate's modes describe how its parameters may be instantiated before and after evaluating it. Different modes may be implemented differently.

- Prolog-style logic programming is 'just' constraint logic programming with only syntactic equality constraints
  - modes should cover other constraints in some meaningful sense, state of the art unclear (to me, anyway)

- Example modes for the equality constraint:
  - binding/assignment: take an uninstantiated parameter on the left and a fully instantiated one on the right
  - equality predicate in most programming languages: is an equality constraint between two ground parameters satisfiable?

Dogma

- No data-structure manipulation except using constraints
  - Makes all manipulation an info effect
- Linearity stops us hiding things
- Search is bad, don't make choices outside constraint solving
  - Even better if the constraint solver makes no choices!
- These three things pretty much guarantee we'll fit on top of a suitable relative of Π

- Prove your type inference works by doing elaboration
  - But that'd take up too much space for today

Constraints for the Simply-Typed Lambda Calculus

$\tau = \tau$	Type equality
$\tau \multimap \langle^{\tau_l}_{\tau_r}$	Type duplication
$x : \tau \in \Gamma$	Binding in context
$\Gamma' := x : \tau; \Gamma$	Context extension
$\Gamma \multimap \langle^{\Gamma_L}_{\Gamma_R}$	Context duplication

Note that the context constraints encode the structural rules. An alternative interpretation could give us a minimal linear calculus.

Some Possible Modes

$$\frac{}{\tau = \tau}$$ Anything subsumed by $\mathit{inout} = \mathit{inout}$

$$\frac{x : \tau \in \Gamma}{\mathit{ground} : \mathit{out} = \mathit{ground}}$$

$$\frac{\Gamma' := x : \tau; \Gamma}{\mathit{out} := \mathit{ground} : \mathit{in}; \mathit{ground}}$$

$$\frac{\Gamma \dashv\langle^{\Gamma_L}_{\Gamma_R}}{\mathit{in} \dashv\langle^{\mathit{out}}_{\mathit{out}}}$$ - duplicate

$$\frac{x : \tau \in \Gamma}{\mathit{ground} : \mathit{inout} = \mathit{inout}}$$

$$\frac{\Gamma' := x : \tau; \Gamma}{\mathit{inout} := \mathit{ground} : \mathit{inout}; \mathit{inout}}$$

$$\frac{\Gamma \dashv\langle^{\Gamma_L}_{\Gamma_R}}{\mathit{out} \dashv\langle^{\mathit{in}}_{\mathit{in}}}$$ - merge

So given a contextless term, we can find the context(s) that make it type.

Tediously Simply-Typed λ-Calculus (unannotated)

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\, Var \qquad \frac{\begin{array}{c}\Gamma_f := x : \tau_p \,;\, \Gamma \\ \Gamma_f \vdash T : \tau_r \\ \tau_f = \tau_p \to \tau_r\end{array}}{\Gamma \vdash \lambda x.T \;:\; \tau_f}\, Lam$$

$$\frac{\begin{array}{c}\Gamma \prec \begin{array}{l}\Gamma_L \\ \Gamma_R\end{array} \\ \Gamma_L \vdash T_f : \tau_f \qquad \Gamma_R \vdash T_p : \tau_p \\ \tau_p \to \tau_r = \tau_f\end{array}}{\Gamma \vdash T_f\ T_p : \tau_r}\, App$$

Introducing Modes - and Eliminating

We can give precise modes to the $=$ constraints in the Lam and App rules:

- Lam's $\tau_f = \tau_p \to \tau_r$ has a mode $\mathit{out} = \mathit{in} \to \mathit{in}$
- App's $\tau_p \to \tau_r = \tau_f$ has a mode $\mathit{in} \to \mathit{out} = \mathit{in}$
- These modes tell us that Lam introduces $\to$ and App eliminates it!
- A similar relationship holds between $\Gamma' := x : \tau; \Gamma$ and $x : \tau \in \Gamma$ - most obvious in a linear calculus, but we can still view Var as eliminating Lam's bindings while App propagates copies

Supporting Annotations

$$\frac{\begin{array}{c}\tau_a \multimap \begin{smallmatrix}\tau_{ap}\\ \tau_{af}\end{smallmatrix} \\ \Gamma_f := x : \tau_{ap} \; ; \; \Gamma \\ \Gamma_f \vdash T : \tau_r \\ \tau_f = \tau_{af} \to \tau_r\end{array}}{\Gamma \vdash \lambda x : \tau_a \; . \; T \; : \; \tau_f}\, ALam$$

- That duplication constraint demonstrates bidirectional dataflow - inwards to the body, outwards to the return type.
- More complex systems exploit this by mixing modes/directions in their typing rules

Implementation?

```
given = (<$)
infixl 3 `given`
type Judgement = Context -> ConstraintGen Type

lam :: Identifier -> Judgement -> Judgement
lam x jb c = withTV (\tf -> withTV (\tp ->
               withTV (\tr -> withCV (\cf ->
                 f tf tp tr cf))))
  where f tf tp tr cf =
    tf `given` cf `ctxtExt` (x, tp) & c *>
               tr <<- jb cf *>
               tf `eqCon` tp :-> tr
```

Implementation Meets Specification

So the Lam rule:

$$\frac{\begin{array}{c}\Gamma_f := x : \tau_p \; ; \Gamma \\ \Gamma_f \vdash T : \tau_r \\ \tau_f = \tau_p \to \tau_r\end{array}}{\Gamma \vdash \lambda x.T \; : \; \tau_f}\, Lam$$

With all the binding noise removed, is implemented by:

```
lam x jb c = ...
  tf `given` cf `ctxtExt` (x, tp) & c *>
              tr <<- jb cf *>
              tf `eqCon` tp :-> tr
```

Constraints for an ML-style System

Variables are now bound to polytypes/type schemes: $\sigma ::= \forall \bar{t}.\tau$

$\tau = \tau$	Monotype equality
$x : \sigma \in \Gamma$	Binding in context
$\Gamma' := x : \sigma; \Gamma$	Context extension
$\sigma \geq \tau$	Type subsumption/instantiation
$\tau \leq_\Gamma \sigma$	Generalisation in context

Instantiation and generalisation have a similar relationship to binding usage and extension. Their satisfaction predicates are n-ary versions of Milner's *Inst* and *Gen* rules.

Not to Over-generalise...

Variable usage and lambdas need to account for polymorphism:

$$\frac{x : \sigma \in \Gamma \qquad \sigma \geq \tau}{\Gamma \vdash x : \tau}\, Var \qquad\qquad \frac{\Gamma_f := x : \forall.\tau_p \; ; \; \Gamma \qquad \Gamma_f \vdash T : \tau_r \qquad \tau_f = \tau_p \rightarrow \tau_r}{\Gamma \vdash \lambda x.T \; : \; \tau_f}\, Lam$$

And this leaves room for let-generalisation to be useful:

$$\frac{\Gamma \dashv\langle{}^{\Gamma_x}_{\Gamma_{temp}} \qquad \Gamma_{temp} \dashv\langle{}^{\Gamma_b}_{\Gamma_{gen}} \qquad \Gamma_x \vdash T_x : \tau_x \qquad \tau_x \leq_{\Gamma_{gen}} \sigma \qquad \Gamma' := x : \sigma; \Gamma_b \qquad \Gamma' \vdash T_b : \tau_b}{\Gamma \vdash \text{let } x = T_x \text{ in } T_b : \tau_b}\, Let$$

I Can't Believe It's Not Hindley-Milner!

9 out of 10 cats couldn't tell this system from Hindley-Milner:

- "Looks like Hindley-Milner to me"
- "Er, yeah, I reckon so?"
- "Mrow!" (tr: more!)

But one asked an important question:

"Wait, does this have principal typings?"

H-M cannot have principal typings. Can 9 out of 10 cats be wrong?

What Is a Universal?

Nothing but a miserable little[1] pile of intersections!

Let's take a constraint store and try some informal rewriting:

$\sigma \geq \tau_1 \ \wedge \ \sigma \geq \tau_2 \ \wedge \ \sigma \geq \tau_3$

To: $\sigma \ \geq (\tau_1 \wedge \tau_2 \wedge \tau_3)$ - but that doesn't entirely make sense.

But $\sigma \ \geq \ (\tau_1 \ \wedge \ \tau_2 \ \wedge \ \tau_3)$ is an entirely valid and even useful subsumption relationship! Why?

There's intersection types hidden in them thar constraints!
Not part of the object language - an *extra-logical* notion?

---

[1] it's only a *countable* infinity

Surveying One's Principalities

- If this system doesn't have principal types, I shall be highly embarrassed.
- If a 'typing' is the constraint generation phase, this system *must* have principal typings.
  - Everything is either syntax-directed or handled by constraints.
  - Let's call this '*principal typings up to constraint solving*'.
- A poor constraint system may yield no desirable properties!
  - Highly non-compositional and counter-intuitive results.
  - I hear some theorem provers are like this...
- Due to the solver's extra-logical intersection types, this system has principal typings *per se*.
  - Must remember to carry solver state in your typing/proof! Or reason effectfully?

The Essence of Module Systems?

Module systems in the ML family seem to be essentially about piping bits of context around to check that bits of term can be plumbed in safely later.

Someone once asked why we can't let modules say *only* what they need from other modules rather than a known signature.

Good question - if I'm part of a team working on something in parallel I might have to work out what my component needs before I know what somebody else's can provide (and maybe give a trivial stopgap for testing)!

Module Systems, Rephrased and Constrained

Suppose we make the module system about constraints on contexts (and the types in them) rather than just contexts?

Our module system can ask questions (eg using intersections) that the core language can't! And we may be able to introduce matching module language constructs where they would be unsafe in a given core language.

Or someone might make the module language a DSL to ensure these constraints were safe within it.

If I had the energy to do a PhD, a quick demo first-order module system would be on the todo list.

Modularity for Free

Even without this, principal typings already give us a degree of modular typechecking.
If we work in this style and have a confluent solver, we're free to check portions of the AST and just keep track of the resulting solver state alongside the types at the 'edges' of our AST fragment. Working on ASTs with holes is not an issue!

Constraints in Context

Something I meant to talk to Adam Gundry about again:

Chunks of context, telescopes etc tend to match up with blocks of metavariables in the constraint store.

More accurately: we could have regioned constraint stores with regions corresponding to parts of the context and thus the source AST.

A variant on the properties given by principal typings - but an important one!

Solving Constrained by Context?

Relatedly: when you translate the *Gen* rule into a Constraint Handling Rule, the guard needs a *non-logical* condition - 'have all the relevant variables been solved for?', or as some would put it, 'is this a forced move yet?'

This breaks standard proof techniques for confluence of CHRs. And we'd like to know we both get an answer and always get the same answer.

Thankfully, we can show this works by constructing a *precongruence* on the variables, showing dataflow limitations with top-level scope at one end and the variables for inner-most blocks forming the leaves.

Only So Congruent

Why a precongruence and not a congruence?

Accepting symmetry would model circular dependencies.
Well-known examples include:

- Failing the occurs check and regularly getting the hump with infinite types
  - OCaml tried it (amongst others). I don't plan on joining them.
- Trying to typecheck polymorphic recursion the unannotated, undecidable way

It seems that here, congruences imply unfortunate infinities.