Université de Montréal

Typer

an ML sibling inheriting from Lisp and Coq

Stefan Monnier

DIRO - Université de Montréal

Emacs vs types

Emacs mostly coded in a form of Lisp

- No compile-time checks
- Soft abstraction boundaries, if any

Type systems

- All about compile-time checks
- Hard abstraction boundaries

How to reconcile the two?

Starting requirements

Implement operations like M_ρ:

- Type level functions

Use HOAS in the CPS of System F:

- Type-level λ abstractions

Avoid duplicating definitions like *Nat*:

- Full dependent types

Full predicate logic

There is more than type-preservation

- To write proofs like *cps_subst_commute*
- Full logic allows Hoare-style full correctness proofs

Research has moved from Haskell to Coq and Agda

- These were designed to write proofs
- Support for *programming* comes second

Why not Coq or Agda?

Not designed as programming languages

Irrelevant/erasable code unclear or poorly controlled

Syntax optimized for use in proofs

Agda lacks macros

Coq's syntax is too complex for my taste

Good macro support requires a fundamentally different syntax

Dependent types

Types can depend on terms; instead of just

$$\mathit{make\_vector} : \alpha \to \mathit{Int} \to \mathit{Vector}\ \alpha$$

We can have

$$\mathit{make\_vector} : \alpha \to (n : \mathit{Int}) \to \mathit{Vector}\ \alpha\ n$$

So $\mathit{make\_vector}\ 0\ (\mathit{fact}\ i)$ has type

$$\mathit{Vector}\ \mathit{Int}\ (\mathit{fact}\ i)$$

Provide a power comparable to predicate logic

Consequences of dependent types

Type-checking requires a form of evaluation:

- Non-termination of terms implies undecidable type-checking
- More serious: it breaks consistency, so can't have proofs

Even "harmless" side-effects are problematic:

- $v_1 = \mathit{make\_vector}\ 0\ (\mathit{random}\ 7); v_2 = \mathit{make\_vector}\ 0\ (\mathit{random}\ 7)$
- Both would have type $\mathit{Vector}\ \alpha\ (\mathit{random}\ 7)$
- Do they really have the same type (and hence, same length)?

$\Rightarrow$ Typer is pure and strongly-normalizing

Side effects and non-termination via *monads*, like in Haskell

Unifying types and terms

Typer is a Pure Type System (PTS):

$$e ::= e_1\ e_2 \mid \text{lambda}\ (x : e_1) \to e_2 \mid x \mid (x : e_1) \to e_2$$

Same core language as proof assistants like Coq and Agda

Identity function could be written:

$$\mathit{id} : (t : \mathit{Type}) \to t \to t;$$

$$\mathit{id} = \text{lambda}\ (t : \mathit{Type}) \to \text{lambda}\ (x : t) \to x;$$

$$\mathit{two} = \mathit{id}\ \mathit{Int}\ 2$$

We use the usual $e_1 \to e_2$ for non-dependent function types

Implicit arguments

Having to write *id Int* 2 is inconvenient

$\Rightarrow$ Typer has *implict* arguments:

$$id : (t : Type) \Rightarrow t \to t;$$

$$id = \text{lambda } (t : Type) \Rightarrow \text{lambda } (x : t) \to x;$$

$$two = id\ 2$$

Any argument can be declared implicit

If Typer can't infer it, you can provide the implicit argument explicitly:

$$two = id\ (t := Int)\ 2$$

Erasable arguments

Even when implict, the *Int* argument is undesirable at run-time

$\Rightarrow$ Typer has *erasable* arguments:

$$id : (t : Type) \Rrightarrow t \to t;$$

$$id = \text{lambda } (t : Type) \Rrightarrow \text{lambda } (x : t) \to x;$$

$$two = id\ 2$$

Erasable arguments are also *implicit*

To be *erasable*, the formal argument can only be used:

- in type annotations
- in an *erasable* actual argument

Datatypes

New types can be defined like in other ML-like languages:

type *Exp*
- | *Num Int*
- | *If Exp Exp Exp*
- | *App Exp Exp*

And they can be checked via the usual pattern matching:

case e
- | *Num* n $\Rightarrow$ ...
- | *If* e_1 e_2 e_3 $\Rightarrow$ ...
- | *App* f *arg* $\Rightarrow$ ...

Equality type

To allow different types for constructors, we have an equality type

$$\begin{array}{l}\text{type } \mathit{Exp}\ \tau \\ \quad \mid \mathit{Num}\ \mathit{Int}\ (\_ :: \tau = \mathit{Int}) \\ \quad \mid \mathit{App}\ (\mathit{Exp}\ (\tau_1 \to \tau))\ (\mathit{Exp}\ \tau_1)\end{array}$$

We use $x :: e$ to indicate an *implicit* argument to a constructor

We use $\_$ to mean that the name is of no importance

The type $\tau = \mathit{Int}$ is a proposition, not a test

The only constructor of this type is *refl*

$$\mathit{refl} : (t : \mathit{Type}) \Rrightarrow (x : t) \Rrightarrow x = x$$

Lisp-style macros

Writing formal proofs is tedious $\Rightarrow$ need automation

Coq has extra 2 sub-languages for that: proof scripts and Ltac

Typer uses Lisp-style macros for the same purpose

- Macros also allow syntax extensions
- Macro calls can occur anywhere, unlike tactic calls
- Macros are useful even if you never write a proof
- Same language to write code and to define “tactics”

Last point important for humans and tools, e.g. debuggers

Infix S-expressions

Source code is first parsed as an S-expression

$$\begin{array}{l}\text{type } \mathit{List}\ \alpha \\ \quad \mid \mathit{Nil} \\ \quad \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)\end{array}$$

is parsed identically to

$$(\mathit{type\_}\ (\_|\_\ (\mathit{List}\ \alpha)\ \mathit{Nil}\ (\mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha))))$$

Underscores in $\mathit{type\_}$ and $\_|\_$ indicate use as prefix/infix

Parsing based on a precedence grammar (currently fixed)

Parsing code as data

S-expressions are just tree *data*

Macros take S-expressions as arguments

Before macro-expansion, semantics is unknown

The parser can't know if the code is actually Typer or some DSL

There are almost no parsing errors for S-expressions:

```
a + ] 5  == (_] (_+ a)) 5
```

Syntax errors only reported when S-exp is "parsed" as Typer code

Really context free syntax

S-expressions are just tree *data*

Infix/prefix/postfix rules do not know context

$$e_1 \to e_2 \to e_3 \quad \equiv \quad e_1 \to (e_2 \to e_3)$$

$$\Downarrow$$

$$\text{lambda } x \to e_1 \to e_2 \quad \equiv \quad \text{lambda } x \to (e_1 \to e_2)$$

Very weak parsing technology

Setting relative precedence of $=$, $\to$, and $:$ was especially delicate

Context-freedom needed because of macros

Parentheses

Parentheses only serve to override precedence

They can be used “anywhere”

$$\begin{aligned}
\text{lambda } x \to e &\equiv \text{lambda } (((x) \to ((e)))) \\
\text{lambda } x : e_1 \to e_2 &\equiv \text{lambda } ((x) : (e_1 \to e_2)) \\
\text{case } e \mid \mathit{Nil} \Rightarrow 0 &\equiv \text{case } e \mid ((\mathit{Nil}) \Rightarrow (0))
\end{aligned}$$

Can’t have syntax like

$$\ldots [\mathit{Int}\ \mathit{String}\ (\mathit{List}\ \mathit{Int})] \ldots$$

because

$$[(\mathit{List}\ \mathit{Int})] \equiv [\mathit{List}\ \mathit{Int}]$$

Macros as proof tactics

Macros receive 2 extra parameters:

- The *goal*: the expected type of the returned code
- The *context*: the list of bound variables with their types

The context includes shadowed, implicit, and erasable variables

Type-checking impossible before macro-expansion

Macro-expansion impossible before type-checking

Elaboration phase combines type-inference and macro-expansion

Types seen by macros can be incomplete

Macros for implicit arguments

Type inference can only guess some implicit arguments

- Associate a macro with a type
- When implicit arg of that type is needed, call the macro

Can be used to implement type classes:

type *Num* α

$\mid$ *NumDict* $(\mathit{add} : \alpha \to \alpha \to \alpha)\ (\mathit{mul} : \alpha \to \alpha \to \alpha)$

provide_implicit *Num find_instance*

Macro *find_instance* looks for appropriate instance in context

Array bounds checking

We can provide safe array access with compile-time bounds checking

$$\textit{vector\_ref} : (\alpha : \textit{Type}) \Rrightarrow (n : \textit{Int}) \Rrightarrow (i : \textit{Int}) \to (i < n) \Rightarrow (\textit{Vector}\ \alpha\ n) \to \alpha$$

The proof of bounds $(i < n)$ is implicit

The $<$ type is associated with macro *omega*

We can always trivially provide the needed proof with a dynamic check

if $i <? n$
then *vector_ref* $i\ v$
else *bounds_error*

Optional arguments

The same mechanism lets us define optional arguments:

$$\text{type } \mathit{Opt}\ \alpha \mid \mathit{Nothing} \mid \mathit{Just}\ \alpha$$

$$\text{provide\_implicit } \mathit{Opt\ Nothing}$$

$$\mathit{find}\ (\mathit{start} :: \mathit{Opt\ Int})\ x\ l = \ldots$$

Can then be called as

$$\mathit{find}\ x_1\ l_1$$

or

$$\mathit{find}\ (\mathit{start} := \mathit{Just}\ 5)\ x_2\ l_2$$

Refinement

Inside the branch of a case we have extra knowledge

The general syntax in Coq is

$$\text{match } e \text{ as .. in .. return .. with } < branches >$$

Typer uses (implicit) equality types instead:

$$\begin{array}{l} \text{case } (e : \mathit{Exp}\ \tau) \\ \quad \mid \mathit{Num}\ n \Rightarrow \ldots \end{array}$$

The context in . . . will include:

- a proof that $\tau = \mathit{Int}$
- a proof that $e = \mathit{Num}\ n$

Simple types stay simple

Typer provides same power as Coq

But it can also be used more naively

It performs type-inference using Hindley-Milner, like ML/Haskell

E.g. you can define

$$\mathit{id}\ x = x$$

$$\begin{array}{l}\mathit{map}\ f\ l = \\ \quad \mathsf{case}\ l \\ \quad\quad \mid \mathit{Nil} \Rightarrow \mathit{Nil} \\ \quad\quad \mid \mathit{Cons}\ x\ l \Rightarrow \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ l)\end{array}$$

All the extra type arguments *will* be inferred

Conclusion

A mostly conservative extension of ML/Haskell

Extensions include

- Fully dependent types
- Powerful macros
- Implicit/optional arguments
- Syntactic extensibility

Compared to ML/Haskell we lose convenient non-obvious termination