A Fully Dependent Assembly Language

Yulong Huang & Jeremy Yallop

University of Cambridge, UK

TYPES 2025, University of Strathclyde, Glasgow

June 2025

Motivation

Dependent types are immediately thrown away after type checking!

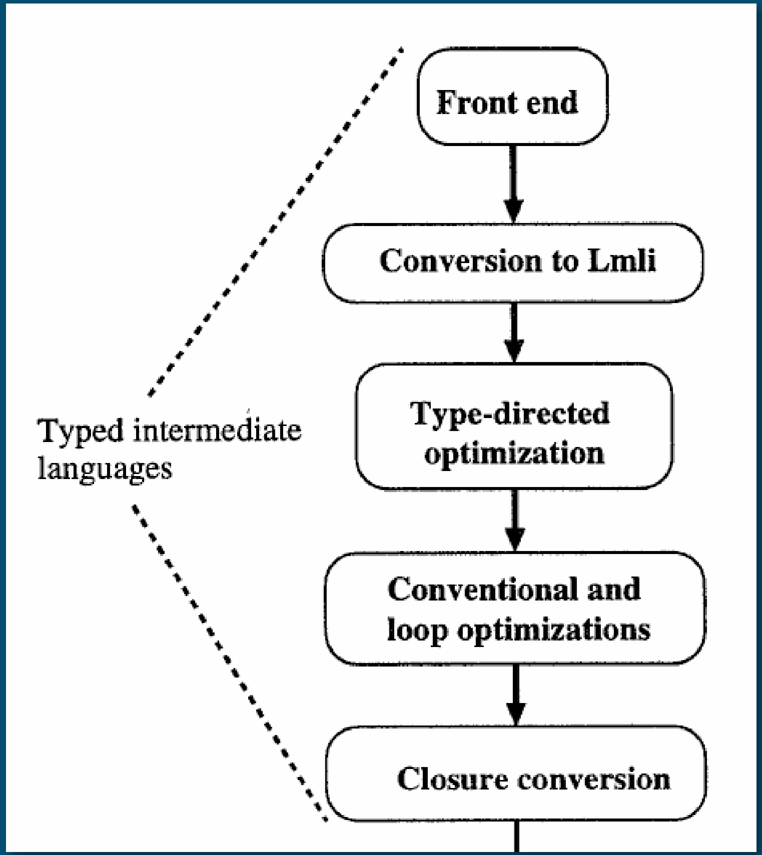
Motivation

Dependent types are immediately thrown away after type checking, but there are good reasons for preserving the types:

Motivation

Dependent types are immediately thrown away after type checking, but there are good reasons for preserving the types:

- To guide compiler transformaions
- To optimize with more information



Tarditi et al. (1996): TIL: A type-directed optimizing compiler for ML

Motivation

Dependent types are immediately thrown away after type checking, but there are good reasons for preserving the types:

- To guide compiler transformaions
- To optimize with more information
- To verify executables through type-checking

```
l_fact:
    code[]{r1:⟨⟩,r2:int,r3:τ_k}.
      bnz r2,l_nonzero
      unpack [α,r3],r3
      ld r4,r3[0]
      ld r1,r3[1]
      mov r2,1
      jmp r4
```

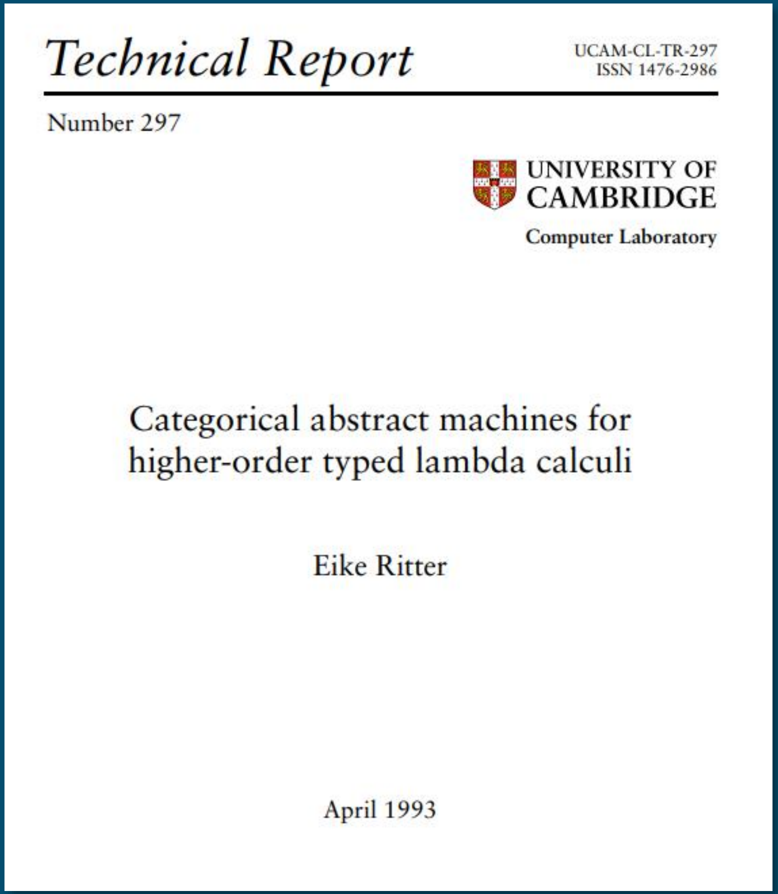
Morrisett et al. (1999): From system F to typed assembly language

Motivation

Dependent types are immediately thrown away after type checking, but there are good reasons for preserving the types:

- To guide compiler transformaions
- To optimize with more information
- To verify executables through type-checking

...and now is the time!



Technical Report

UCAM-CL-TR-297
ISSN 1476-2986

Number 297

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Categorical abstract machines for higher-order typed lambda calculi

Eike Ritter

April 1993

Ritter (1993): Categorical abstract syntax for higher-order typed lambda calculi

Dependent assembly how?

One should specify:

- how types depend on instructions
- the equational theory of instructions
- what is a dependent stack…

Dependent assembly how?

~~One should specify:~~

- ~~how types depend on instructions~~
- ~~the equational theory of instructions~~
- ~~what is a dependent stack...~~

One should separate high-level dependent types and low-level assembly code.

Dependent assembly: Syntax

Assembly: instruction set for a stack machine

$$I \;::=\; \texttt{LIT}\ c \mid \texttt{POP} \mid \texttt{VAR}\ x \mid \texttt{CLO}\ n\ lab \mid \texttt{APP} \mid I\ ;\ I' \mid \ldots$$

Term calculus: fully dependently typed calculus for specifying types of assembly code

$$e,\ A \;::=\; x \mid e\ e' \mid lab\ \{e_1, \ldots, e_n\} \mid \Pi x{:}A.B \mid U \mid \ldots$$

Term calculus: Defunctionalized CC

The term calculus is similar to calculus of constructions (CC) except:

- there is no lambda abstraction
- context contains a fixed set of function labels
- labels form closures with lists of terms

The defunctionalized CC is consistent and the function labels can be generated from a source program in CC.

$$\frac{\mathit{\Gamma} \vdash e_1, \ldots, e_n : \mathit{\Delta} \qquad lab(\mathit{\Delta},\, x{:}A \mapsto e : B) \in \mathit{\Gamma}}{\mathit{\Gamma} \vdash lab\{e_1, \ldots, e_n\} : (\mathit{\Pi} x{:}A.B)[e_1, \ldots, e_n \,/\, \mathit{\Delta}]}$$

Huang and Yallop (2023): Defunctionalization with dependent types

Assembly: SECD machine

Runtime values are closed values in the term calculus (i.e. closures, types, base values).

$$v \ ::= \ lab\,\{v_1,\ \ldots,\ v_n\} \mid A \mid \underline{b} \mid \ldots$$

A machine state $<I,\ Env,\ St,\ Fr>_P$ is made of:

- An instruction sequence (control) $\quad I$
- A runtime environment of values $\quad Env \ : \ List\ v$
- A stack of values $\quad St \ : \ List\ v$
- A stack of call frames (dump) $\quad Fr \ : \ List\,(I \times Env \times St)$
- A list of procedures $\quad P \ : \ Label \longrightarrow I$

Machine step: $\boxed{<I,\ Env,\ St,\ Fr>_P \ \longrightarrow\ <I',\ Env',\ St',\ Fr'>_P}$

Typing the assembly (judgement)

An abstract stack σ is a list of terms in the defunctionalized CC.

The typing judgement

$$\boxed{\Gamma \ \vdash I : \sigma \longrightarrow \sigma'}$$

says that instruction I transforms stack σ to stack σ', modelling the computation like an abstract interpreter.

Typing the assembly (basic operations)

$$\frac{x : A \in \Gamma}{\Gamma \vdash \texttt{VAR}\ x : \sigma \longrightarrow \sigma :: x}$$

$$< \texttt{VAR}\ x\ ;\ I,\ Env,\ St \qquad ,\ Fr>_P \longrightarrow$$
$$< I \qquad\qquad ,\ Env,\ St :: v,\ Fr>_P$$
$$\text{where } v = Env(x)$$

$$\frac{}{\Gamma \vdash \texttt{POP}\ : \sigma,\ t \longrightarrow \sigma}$$

$$< \texttt{POP}\ ;\ I,\quad Env,\ St :: v,\ Fr>_P \longrightarrow$$
$$< I \qquad ,\quad Env,\ St \qquad ,\ Fr>_P$$

Typing the assembly (closure)

`Clo` n lab forms a closure with the top n items on the stack.

$$\frac{\Gamma \vdash e_1, \ldots, e_n : \Delta \qquad lab(\Delta,\ x{:}A \mapsto e : B) \in \Gamma}{\Gamma \vdash \texttt{CLO}\ n\ lab : \sigma :: e_1 :: \ldots :: e_n \to \sigma :: lab\{e_1, \ldots, e_n\}}$$

$$\begin{aligned} &< \texttt{CLO}\ n\ lab\ ;\ I,\ Env,\ St :: v_1 :: \ldots :: v_n\ ,\ Fr >_P \longrightarrow \\ &< I\ ,\ Env,\ St :: lab\{v_1, \ldots, v_n\}\ ,\ Fr >_P \end{aligned}$$

Typing the assembly (application)

Application: loads instructions according to lab, fills in the environment, saves current (I, Env, St) on a new call frame.

$$\frac{\Gamma \vdash e : \Pi x{:}A.B \qquad \Gamma \vdash e' : A}{\Gamma \vdash \mathtt{APP} : \sigma :: e :: e' \;\rightarrow\; \sigma :: e\ e'}$$

$$\begin{array}{l} < \mathtt{APP}\ ;\ I,\ Env \qquad ,\ St :: lab\{Env'\} :: v,\ Fr >_P \longrightarrow \\ < P(lab)\ ,\ Env' :: v,\ [] \qquad ,\ Fr :: (I,\ Env,\ St)\ >_P \end{array}$$

Compilation

Now, we can define a simple compilation function that generates dependent assembly code from defunctionalized CC code:

$$
\begin{array}{lcl}
\mathsf{cp}\ \ x & = & \mathtt{VAR}\ x \\
\mathsf{cp}\ \ \mathit{\Pi x{:}A.B} & = & \mathtt{LIT}\ \mathit{\Pi x{:}A.B} \\
\mathsf{cp}\ \ U & = & \mathtt{LIT}\ U \\
\mathsf{cp}\ \ \mathit{lab}\ \{e_1,\ \ldots,\ e_n\} & = & \mathsf{cp}\ e_1\ ;\ \ldots\ ;\ \mathsf{cp}\ e_n\ ;\ \mathtt{CLO}\ n\ \mathit{lab} \\
\mathsf{cp}\ \ e\ e' & = & \mathsf{cp}\ e\ ;\ \mathsf{cp}\ e'\ ;\ \mathtt{APP}
\end{array}
$$

Correctness of compilation

Type preservation: $\Gamma \vdash e : A \implies \Gamma \vdash I : \sigma \to \sigma :: e$ *for all* σ.

Correctness (WIP): *For all base types* A, *if* $\cdot \vdash e : A$ *and* $e \rightsquigarrow^* v$, *then*

$$< \mathsf{cp}\ e, [], [], [] >_P \longrightarrow < [], [], [] :: v, [] >_P$$

Typing the machine states

Runtime values can be typed since they are closed values in the term calculus.

$\vdash v : A$

Other components of the machine state are also typable:

$\vdash Env : \Gamma$	env implements a context of type Γ
$\vdash_{Env} St : \sigma$	st implements σ w.r.t. well-formed Env

(judgements omitted for frames and procedures)

Above combine to a well-formedness judgement for machine states:

$\vdash < I, Env, St, Fr >_P$

Type safety

Progress:

$$If\ \vdash\ <I,\ Env,\ St,\ Fr>_P\ then\ <I,\ Env,\ St,\ Fr>_P\ \longrightarrow\ <I',\ Env',\ St',\ Fr'>_P$$

Preservation:

$$If\ \vdash\ <I,\ Env,\ St,\ Fr>_P\ and\ <I,\ Env,\ St,\ Fr>_P\ \longrightarrow\ <I',\ Env',\ St',\ Fr'>_P$$
$$then\ \vdash\ <I',\ Env',\ St',\ Fr'>_P$$

Future directions

- Termination
- Agda formalization of meta-theory
- Erasure of runtime types
- Certified optimization
- Datatypes and runtime representations
- Quantitative types for better erasure and linearity

Thank you!

…and questions?

Speaker email: yh419@cam.ac.uk