A Fully Dependent Assembly Language

Yulong Huang and Jeremy Yallop

University of Cambridge, UK

We report our progress in developing an assembly language with fully dependent types to support type-preserving compilation of dependent type theory, and thus improving the compilation of existing systems which erase types at an early stage. In this abstract, we outline the fundamental design principles of the language and present our encoding of it in Agda. [1]

A type-preserving compiler [8] transforms the source code into a strongly typed assembly language with a series of typed compiler transformations, preserving types through all phases. Many optimizations benefit from type information. With dependent types, type signatures can contain code specifications and type-checking the assembly language verifies that the specifications are met, essentially implementing proof-carrying code [9].

Earlier attempts towards dependent assembly languages, such as Ritter's categorical abstract machine [10] and Winwood's Singleton [11], are incompatible with recent advances in dependently-typed transformations like CPS [3], ANF [6], closure conversion [2], and defunctionalization [4]. A desirable target language should both accommodate fully dependent types and be able to compile from the intermediate representations of previous transformations.

We propose a two-level design. The syntax of the typed assembly language consists of a set of instructions of a stack machine (I) and a dependently typed calculus of specifications (e, A).

$$I ::= \texttt{LIT}\ c \mid \texttt{APP} \mid \texttt{CLO}\ n\ \mathit{lab} \mid \texttt{POP} \mid I;I' \mid \ldots$$
$$e, A ::= x \mid e\,e' \mid \mathit{lab}\{e_1,\ \ldots,\ e_n\} \mid \Pi x{:}A.B \mid U \mid \ldots$$

The type judgement for each instruction *models* the computation like an abstract interpreter, in the form of $\Gamma \vdash I : \sigma \to \sigma'$. A stack σ is a list of specification terms. The judgement states that instruction I transforms stack σ to σ', similar to the conventional stack typing (e.g. WebAssembly [1]), but it keeps track of the stack's *content* instead of the types of the contents. For example, the rule for pushing a literal constant c in our language is $\Gamma \vdash \texttt{LIT}\ c : \sigma \to \sigma :: c$.

Tracking stack contents is necessary to support fully dependent types (since computation happens at type level), and separating specification from instructions avoids problems such as the need to check equality of instruction sequences during type checking. Function application, which is complicated by type dependency, is simply described by an application in the specification calculus (shown at the left below, note that B does not appear in the specification).

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

The specification language is a *defunctionalized* calculus, an intermediate representation from dependently typed defunctionalization [4]. It has no lambda abstractions and no way to create new functions. We can only create a closure of type $\Pi x{:}A.B$ by paring a labelled code lab (from a fixed set of labels, defined in the context Γ) with a list of terms $e_1,\ \ldots,\ e_n$ (of types Δ) instantiating the free variables in the code. In assembly, it is reflected by an operation $\texttt{CLO}\ n\ \mathit{lab}$ that takes the top n elements of the stack and forms a closure object with the label lab on top of the stack, as shown in the judgement at the right above.

The specification calculus is shown to be consistent [4]. We need to prove type safety for the assembly, in other words, that the typing rule's abstract interpretation correctly models runtime behaviour.

[1] See `https://github.com/H-Yulong/ShallowStack`.

We are in the process of encoding the assembly language in Agda to formally verify its meta-theory. The encoding is challenging: besides the classic problem of encoding dependent type theory in itself, we also have to find a suitable representation for labelled defunctionalized code.

We use shallow embedding [7, 5] to encode syntax, which coincides the definitional equality of the object theory and the host language to avoid the problem of having tedious equality conversions (known as "transport hell") — every equation in the object theory is trivially resolved to `refl` in the host.

Defunctionalization is trickier. As we have previously observed [4], the usual method of representing each function label with a constructor of a GADT fails to extend to inductive families, because it requires the data type to be indexed by itself (hence failing positivity checks), and the interpretation function for code labels is not obviously terminating. [2]

We have found a workaround for the positivity checks to define an indexed family `Pi` over the shallow-embedded syntax for defunctionalized code and an interpretation function `interp` that passes Agda's termination checker, and we will present the techniques used in the talk. Below shows the (simplified) type signatures of `Pi` and `interp`, and code labels for two functions. [3]

```
data Pi : (id : ℕ) (Γ : Con) (A : Ty Γ) (B : Ty (Γ , A)) → Set where
  CNat : Pi 0 · Nat U · ⊢ λx:Nat.Nat
  App  : Pi 0 (·, U, Π 0 U, Π 1 (app 1 0)) 2 (app 2 0) A : U, B : ΠA:U.U, f : Πx:A.B ⊢ λx:A.f x
interp : Pi id Γ A B → Tm (Γ , A) B
```

The intrinsic typing rules of the stack machine are straightforward to express with shallow-embedded syntax and defunctionalization. The rules for `POP` and `APP` are exactly like their typing judgements. Instruction `CLO` n *lab* takes an instance argument `pf` which proves that the first n items on the stack have types Δ, and Agda can find this proof automatically. We also have rules for creating and eliminating base types such as booleans and natural numbers.

```
data Instr (Γ : Con) : Stack Γ m → Stack Γ n → Set where
  LIT : (n : ℕ) → Instr Γ σ (σ :: nat n)
  APP : {f : Tm Γ (Π A B)} {a : Tm Γ A} → Instr Γ (σ :: f :: a) (σ :: app f a)
  CLO : (n : ℕ)(lab : Pi id Δ A B)
        {{ pf : Γ ⊢ (take n σ) of Δ }} → Instr Γ (drop n σ :: lab [[ pf ]]s)
```

Here is a type-checked example of an instruction sequence that computes 5 via `App` and `Add` (which corresponds to $\lambda y{:}Nat.x+y$). Firstly, it creates a closure for `App`, instantiating the free variables to Nat, `CNat`$\{\}$, and `Add`$\{2\}$, then applies the closure to 3, eventually computing $2+3$. The assembly code type-checks because $2+3=5$, which Agda can realize without proofs.

```
code : Is · · (· :: nat 5)          Abstract stack content
code = TLIT Nat                      Nat
    >> CLO 0 CNat                    Nat :: CNat{}
    >> LIT 2                         Nat :: CNat{} :: 2
    >> CLO 1 Add                     Nat :: CNat{} :: Add{2}
    >> CLO 3 App                     App{Nat, CNat{}, Add{2}}
    >> LIT 3                         App{Nat, CNat{}, Add{2}} :: 3
    >> APP                           5
    >> RET                           5
```

Our dependent assembly language uses a two-level design that leaves the machine's model simple, but maintains a rich and expressive type system to accommodate type-preserving compilation. We plan to use this Agda formalization to investigate type erasure, assembly code generation and optimization, and incorporating quantitative type theory in the future.

[2]We use type-in-type in this abstract and omitted a problem with universe levels for simplicity.

[3]We use de-Bruijn indices (bold numbers) for variables.

References

[1] WebAssembly Core Specification.

[2] William J. Bowman and Amal Ahmed. Typed closure conversion for the calculus of constructions. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 797–811. ACM, 2018.

[3] William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. Type-preserving CPS translation of Σ and Π types is not not possible. *Proc. ACM Program. Lang.*, 2(POPL):22:1–22:33, 2018.

[4] Yulong Huang and Jeremy Yallop. Defunctionalization with dependent types. *Proc. ACM Program. Lang.*, 7(PLDI):516–538, 2023.

[5] Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow embedding of type theory is morally correct. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 329–365. Springer, 2019.

[6] Paulette Koronkevich, Ramon Rakow, Amal Ahmed, and William J. Bowman. ANF preserves dependent types up to extensional equality. *J. Funct. Program.*, 32:e12, 2022.

[7] Conor McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In Bruno C. d. S. Oliveira and Marcin Zalewski, editors, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010*, pages 1–12. ACM, 2010.

[8] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 85–97. ACM, 1998.

[9] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997.

[10] Eike Ritter. Categorical abstract machines for higher-order typed lambda-calculi. *Theor. Comput. Sci.*, 136(1):125–162, 1994.

[11] Simon Winwood and Manuel M. T. Chakravarty. Singleton: a general-purpose dependently-typed assembly language. In Stephanie Weirich and Derek Dreyer, editors, *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*, pages 3–14. ACM, 2011.