Fully Dependent Abstract Machines

For type-preserving compilation of dependent type theory

YULONG HUANG, University of Cambridge, United Kingdom
JEREMY YALLOP, University of Cambridge, United Kingdom

A dependently typed low-level target language can provide strong guarantees about compiled code, specifying both the types of program fragments and the exact values they compute. Although recent advances in type-preserving compilation of dependently typed languages have yielded a variety of intermediate languages for closure conversion, CPS, defunctionalization, and A-normal form, there is no low-level target language to date that supports fully dependent types with large elimination.

We introduce a principle for developing dependently typed abstract machines in which instructions are typed as transformers of abstract stacks of terms drawn from a specification calculus. We apply the principle to develop two machines — the dependent SECD machine and the TAL-style dependent abstract machine (DAM) — that enjoy *total correctness*: every well-typed machine halts with exactly the value specified by its type. Finally, we give a type-preserving translation targeting DAM which, combined with existing work on defunctionalization, yields the first type-preserving compiler for a fully dependently typed language, with whole-program and separate compilation correctness checkable by type checking the compiled instructions, independently of the compiler or source code that produced them.

CCS Concepts: • **Theory of computation** → **Type theory**; • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: type-preserving compilation, abstract machines, dependent types

1 Introduction

It is convenient to assign types to stack machine instructions. For example, consider the instruction

$$\mathsf{app} \;:\; \Delta :: (A \to B) :: A \;\longrightarrow\; \Delta :: B$$

whose type indicates that it applies the second-to-top item, a closure of type $A \to B$, to the top item of type A, leaving the result of type B on the remaining stack (of type Δ). If the closure has a dependent type, $\Pi x{:}A.\,B$, a difficulty arises: the result type $B[v/x]$ now depends on the value of the top stack item v, but the syntax only provides access to the item's type A. To assign computation dependent types, the instruction's type must track the values on the stack and how they change:

$$\mathsf{app} \;:\; (\sigma : \Delta) :: (f : \Pi x{:}A.\,B) :: (v : A) \;\longrightarrow\; (\sigma : \Delta) :: (f\ v : B[v/x])$$

where (σ, f, v) are expressions of a dependently typed calculus. We show that this way of typing instructions leads to dependently typed abstract machines, which are suitable targets for *type-preserving compilation*.

Preserving types through compilation has a number of advantages: it allows compilers to perform type-directed program optimization [Tarditi et al. 1996], analysis [Jagannathan et al. 1997], and elaboration [Kovács 2020; Wadler and Blott 1989], supports safe linking of separately compiled programs [Glew and Morrisett 1999], and increases confidence in compilation correctness [Patrignani et al. 2019]. Dependently typed programming languages can specify precise properties of programs, making them ideal for proof assistants like Agda [Bove et al. 2009], Rocq [Rocq Development Team 2025], Idris [Brady 2021], and Lean [de Moura and Ullrich 2021]. The implementations of these systems can verify the specifications with type checking, but they subsequently all discard types at an early stage of compilation. The safety guarantees provided by dependent types are consequently unavailable in generated code and the rich type information in the source code is not fully utilized for optimizations.

Authors' Contact Information: Yulong Huang, University of Cambridge, United Kingdom, yh419@cam.ac.uk; Jeremy Yallop, University of Cambridge, United Kingdom, jeremy.yallop@cl.cam.ac.uk.

Preserving dependent types through compilation is a challenging problem. Conventional compiler transformations do not extend straightforwardly to dependent types due to type mismatches and usages of logically-inconsistent features [Bowman and Ahmed 2018; Huang and Yallop 2023a], and in some cases, extension is considered impossible [Barthe and Uustalu 2002]. Challenges also arise from proving the transformations type-preserving and correct: the mutual-inductively defined judgements used for dependent types can introduce circularity problems [Bowman 2019] and in some cases, extensional type theory is required to establish correctness [Koronkevich et al. 2022].

Overcoming these difficulties promises significant rewards. Since preserving type equivalence requires preserving term equivalence, the semantics of the source language are preserved along with the types. Generated code can be type-checked to verify that correctness properties still hold, making dependently typed assembly language a kind of *proof carrying code* [Necula 1997]. Preserving types allows linking to check the compatibility of compiled modules, so that the guarantees available to monolithic dependently typed programs remain available with separate compilation. A sophisticated compiler might also perform bound checking elimination [Xi and Harper 2001] data layout optimisation [Weeks 2006], and other type-directed analyses and optimizations used in simply typed systems.

Several recent advances have adapted conventional compiler transformations such as continuation-passing style transformation [Bowman et al. 2018], closure conversion [Bowman and Ahmed 2018], defunctionalization [Huang and Yallop 2023a], and conversion to A-normal form [Koronkevich et al. 2022]. A key missing requirement is a target language for type-preserving compilers, a low-level language with dependent types and compatible with translations from the output languages of these transformations. None of the existing work meets the need: it either implements only a restricted version of dependent types [e.g. Xi and Harper 2001] or is incompatible with dependently typed program transformations [e.g. Winwood and Chakravarty 2011].

This work provides the missing piece. We introduce a typing principle that directly models stack contents in an abstract interpretation style, relating runtime values to compile-time terms. We use a two-level syntax: a dependently typed calculus for specifying the types and terms for compile-time reasoning and a low-level instruction set for manipulating runtime values. We derive two abstract machines from our principle, which serve as vehicles for our technical contributions:

- First (§2) we develop a dependently typed SECD machine based on the calculus of constructions [Coquand and Huet 1988], and establish its metatheoretical properties: *type safety* via progress and preservation (§2.5); *partial correctness*, a stronger property that says that if a well-formed instruction sequence halts, it does so with the correct value; *termination* (§2.6) and *total correctness*, which says that a well-formed instruction sequence always halts with the correct value. These properties serve as a basis for both whole-program and separate compilation correctness (§2.7), which can be checked independently of the compiler and source program used to generate the instructions.
- Second (§3), we apply the same principle to the *Defunctionalized* Calculus of Constructions (DCC), to develop a more practical dependent abstract machine (DAM) that features fixed code blocks (rather than code kept on the stack), and show its termination and total correctness using a backward translation to the machine of §2. We combine our work with DCC (§3.6) to complete a type-preserving compiler for a fully dependently typed language with large elimination, translating programs from the calculus of constructions to a practical abstract machine.

Finally, §4 offers closing discussion and §4.1 surveys related work.

2 Dependent SECD Machine

In this section, we apply the principle of modelling stack contents to assign dependent types to the SECD machine [Landin 1964]. We first introduce the dependently typed specification calculus CC_ω (§2.1), then give the syntax and operational semantics of the SECD machine (§2.2), and assign types to its instructions (§2.3) and machine configurations (§2.4). We show that well-typed SECD machines have type safety (§2.5) and termination (§2.6), and combining these gives a *total correctness* property (Corollary 2.23). Finally, we define a compilation function from CC_ω to SECD instructions (§2.7) and illustrate that type preservation supports correct separate compilation (Corollary 2.27).

2.1 Specification Language: The Calculus of Constructions

Our specification language CC_ω is a variation of the Calculus of Constructions [Coquand and Huet 1988] with dependent pairs, unit type, and natural numbers, extended with a predicative universe hierarchy in the style of Luo [1989], with typed reductions and conversions. CC_ω is concise and expressive: it is a minimal setup that supports *large elimination*, i.e. creating types by iterating over small datatypes such as natural numbers. Large elimination makes the calculus *fully dependently typed*, meaning that the dependency of types on values cannot be erased (e.g. by translating to System F_ω [Geuvers 1994]). We include large elimination to show that our work can be generalized to full-scale dependently typed systems, where large elimination is a common feature.

Fig. 1 shows the syntax of CC_ω. Formally, the variables are de Bruijn indices, but we present them in a named-variable style for readability.

CC_ω

Expressions	A, B, M, N, P	::=	$x \mid U_i \mid \Pi x{:}A.\,B \mid M\ N \mid \lambda x{:}A.\,M$
			$\mid \Sigma x{:}A.\,B \mid \langle M, N\rangle \mid \mathrm{fst}\ M \mid \mathrm{snd}\ M$
			$\mid \mathrm{Unit} \mid \langle\rangle \mid \mathrm{Nat} \mid \mathrm{zero} \mid \mathrm{suc}\ M \mid \mathrm{rec}_P(M, (x, p).M', N)$
Contexts	Γ	::=	$\cdot \mid \Gamma, x{:}A$

Fig. 1. Syntax of CC_ω

Types and universes. We use a *predicative hierarchy* of universes $\{U_i\}_{i\in\mathbb{N}}$ with the type of U_i being U_{i+1} (TY-UNIVERSE), following Martin-Löf [1984] and many dependently typed systems [Bove et al. 2009; Rocq Development Team 2025], to prevent the paradoxical construction that arises when a single sort is used as the type of all types [Coquand 1986].

Typing rules have the form $\Gamma \vdash M : A$, which indicates that M has the type A under the context Γ, where contexts associate types with variables. Types and terms are not distinguished syntactically. If $\Gamma \vdash A : U_i$ (for some i), then A is a well-formed type in context Γ; we write $\Gamma \vdash A : U$, omitting the i subscript as a convention. Base types like Unit and Nat are in U_0. Both Π-types and Σ-types inhabit the larger of the two universes inhabited by their constituents A and B (TY-PI, TY-SIGMA).

Introductions of base types, base values, variables, and universes (the base cases of typing judgements) are valid only if the telescope context Γ is well-formed (denoted as $\vdash \Gamma$), i.e. if all types in Γ are well-formed.

CC_ω

$$\textsc{ty-Universe}\ \frac{\vdash \Gamma}{\Gamma \vdash U_i : U_{i+1}} \qquad \textsc{ty-Pi}\ \frac{\Gamma \vdash A : U_i \quad \Gamma, x{:}A \vdash B : U_j}{\Gamma \vdash \Pi x{:}A.\,B : U_{\max(i,j)}} \qquad \textsc{ty-Sigma}\ \frac{\Gamma \vdash A : U_i \quad \Gamma, x{:}A \vdash B : U_j}{\Gamma \vdash \Sigma x{:}A.\,B : U_{\max(i,j)}}$$

Dependent functions and pairs. The typing rules for dependent functions and pairs are standard. The λ-abstraction $\lambda x{:}A.\, M$ creates a dependent function of type $\Pi x{:}A.\, B$ (TY-LAMBDA), and the input is substituted in B for x in the result type of an application (TY-APPLY).

In a dependent pair $\langle M, N\rangle$, the type B of the second component N may depend on the first component M (named x in the type), so that the type of N is B with M substituted for x (TY-PAIR). Dependent pairs are eliminated with projections, where the type of the second projection depends on the first projection (TY-SND). As conventions, we write $A \to B$ and $A \times B$ for non-dependent functions and pairs (when x is not free in B).

CC_ω

$$\textsc{ty-Lambda}\ \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : \Pi x{:}A.\, B} \qquad \textsc{ty-Apply}\ \frac{\Gamma \vdash M : \Pi x{:}A.\, B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}$$

$$\textsc{ty-Pair}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \langle M, N\rangle : \Sigma x{:}A.\, B} \qquad \textsc{ty-Fst}\ \frac{\Gamma \vdash M : \Sigma x{:}A.\, B}{\Gamma \vdash \mathrm{fst}\ M : A} \qquad \textsc{ty-Snd}\ \frac{\Gamma \vdash M : \Sigma x{:}A.\, B}{\Gamma \vdash \mathrm{snd}\ M : B[\mathrm{fst}\ M/x]}$$

Reduction and definitional equality. We present CC_ω with typed reductions of the form $\Gamma \vdash M \triangleright N : A$, with β-reduction for function applications, projections for pairs, and the two cases for the recursor (Fig. 2a). $\Gamma \vdash M \triangleright^* N : A$ stands for the reflexive-transitive closure of reductions congruenced under Π, Σ, applications, etc. Typed reductions give subject reduction immediately and avoids the circular argument involved in untyped reduction: to prove subject reduction, we need to prove the injectivity of Π-types that relies on a notion of typed confluence which requires subject reduction again [Goguen 1994].

The definitional equality $\Gamma \vdash M \equiv N : A$ is also presented as a typed relation. Two terms are equivalent if they reduce to a same term (EQ-REDUCE), or are η-equivalent functions or pairs (EQ-ETAPI, EQ-ETASIGMA). The conversion rule TY-EQUIV transports equivalent types in typing judgements.

CC_ω

$$\textsc{eq-Reduce}\ \frac{\Gamma \vdash M \triangleright^* M' : A \quad \Gamma \vdash N \triangleright^* M' : A}{\Gamma \vdash M \equiv N : A} \qquad \textsc{ty-Equiv}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash A \equiv B : U}{\Gamma \vdash M : B}$$

Natural numbers. The natural number type has the usual constructors zero and suc. Its eliminator $\mathrm{rec}_P(M, (x, p).M', N)$ (also known as the recursor) takes a predicate on natural numbers P, a zero case M, an inductive case M' (with bound variables x, p), and a natural-number scrutinee N, and produces a term of type $P[N/x]$ (TY-REC). The recursor evaluates to the base case M when $N = \mathrm{zero}$. When $N = \mathrm{suc}\ N'$, it reduces to the inductive case M' with x and p substituted with the predecessor N' and the result of recursing over N', i.e. $\mathrm{rec}_P(M, (x, p).M', N')$. Note that the recursor supports large elimination: we could produce types by recursing over a natural number.

CC_ω

$$\textsc{ty-Rec}\ \frac{\begin{array}{c}\Gamma, x{:}\mathrm{Nat} \vdash P : U \quad \Gamma \vdash N : \mathrm{Nat} \\ \Gamma \vdash M : P[\mathrm{zero}/x] \quad \Gamma, x{:}\mathrm{Nat}, p{:}P \vdash M' : P[\mathrm{suc}\ x/x]\end{array}}{\Gamma \vdash \mathrm{rec}_P(M, (x, p).M', N) : P[N/x]}$$

Properties of CC_ω. We now state some important properties of CC_ω. Structurally, the judgements of CC_ω have well-formed parameters, for example, if a judgement $\Gamma \vdash M : A$ holds, then Γ

is a well-formed context and A is a well-formed type under Γ, and similarly for reduction and equivalence judgements. Meta-theoretically, CC_ω has deep normalization (Theorem 2.4) and is consistent (Theorem 2.5), with injective type formers (Lemma 2.2), and different type formers are never definitionally equal (Lemma 2.3).

These properties give CC_ω decidable type-checking, as Danielsson et al. [2026] have recently mechanized. We could follow Danielsson et al. and present our calculus in Martin-Löf type theory with separate judgements for well-formed types, type equivalence, etc., but we favour the concise Calculus of Constructions style for simplicity of presentation. The exact choice of the specification calculus does not matter, as long as it is normalizing, consistent, and has injectivity and non-confusion for type formers which are required to establish type safety for the SECD machine (see §2.5).

Lemma 2.1 (Well-formedness of typing judgements). *The parameters of the judgements are well-formed in the following sense:*

- *If* $\Gamma \vdash M : A$, *then* $\vdash \Gamma$ *and* $\Gamma \vdash A : U$.
- *If* $\Gamma \vdash M \triangleright N : A$, *then* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$.
- *If* $\Gamma \vdash M \equiv N : A$, *then* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$.

Lemma 2.2 (Injectivity of type formers). *The type formers are injective in the following sense:*

- *If* $\Gamma \vdash U_i \equiv U_j : U$, *then* $i = j$.
- *If* $\Gamma \vdash (\Pi x{:}A.\, B) \equiv (\Pi x{:}A'.\, B') : U$, *then* $\Gamma \vdash A \equiv A' : U$ *and* $\Gamma, x{:}A \vdash B\ x \equiv B'\ x : U$.
- *If* $\Gamma \vdash (\Sigma x{:}A.\, B) \equiv (\Sigma x{:}A'.\, B') : U$, *then* $\Gamma \vdash A \equiv A' : U$ *and* $\Gamma, x{:}A \vdash B\ x \equiv B'\ x : U$.

Lemma 2.3 (Non-confusion of type formers). *Different type formers cannot be equivalent. Specifically,* $\Gamma \vdash U_i \equiv \mathit{Nat} : U$ *is not derivable,* $\Gamma \vdash U_i \equiv \mathit{Unit} : U$ *is not derivable, and so on.*

Theorem 2.4 (Deep normalization). *If* $\Gamma \vdash M : A$, *then* $\Gamma \vdash M \equiv N : A$ *for some* N *in* β*-short-*η*-long normal form.*

Theorem 2.5 (Consistency). *There is no* M *such that* $\vdash M : (\Pi A{:}U_0.\, A)$.

2.2 SECD Machine

An SECD machine [Landin 1964] consists of a stack, a runtime environment, a sequence of instructions (code), and a stack of call frames (originally called dumps). Fig. 3 shows the syntax of the SECD machine we consider. The instruction set manipulates the machine's runtime values, which include natural numbers, the unit value, pairs, closed CC_ω types A, and code closures $\langle \mathsf{env}, \mathsf{ins} \rangle$. The runtime environments and stacks are lists of values. A call frame is a list of triples $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st} \rangle$ that represents the environment, stack, and the rest of instructions that were saved during function calls.

The machine's operational semantics is defined in Fig. 4 as a reduction relation $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \longrightarrow \langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}' \rangle$. Most instructions operate straightforwardly within the current call frame, e.g. `var` x looks up the value at position x in the runtime environment[1] and copies it to the stack, `ty` A pushes a CC_ω type onto the stack and closes it with the current environment, etc.

Evaluating function calls. Closures $\langle \mathsf{env}, \mathsf{ins} \rangle$ are created by `pushc` ins that pairs the instruction with the current environment (SECD-RED-PUSHC) or by `clo` n ins that takes the top n items from the stack and forms a closure with the instruction (SECD-RED-CLO). Upon application, if the stack's top two elements are a closure $\langle \mathsf{env}', \mathsf{ins}' \rangle$ and a value v, then the machine saves the caller frame's

[1] Runtime environments do not need to store variable names since the variables are de Bruijn indices.

CC_ω

$\boxed{\Gamma \vdash M \triangleright N : A}$ *(Reduction)*

RED-BETA
$$\frac{\Gamma, x{:}A \vdash N : B \quad \Gamma \vdash M : A}{\Gamma \vdash (\lambda x{:}A.\, N)\ M \triangleright N[M/x] : B[M/x]}$$

RED-FST
$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \mathrm{fst}\ \langle M, N\rangle \triangleright M : A}$$

RED-SND
$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \mathrm{snd}\ \langle M, N\rangle \triangleright N : B[M/x]}$$

RED-RECZ
$$\frac{\Gamma, x{:}\mathrm{Nat} \vdash P : U \quad \Gamma \vdash M : P[\mathrm{zero}/x] \quad \Gamma, x{:}\mathrm{Nat}, p{:}P \vdash M' : P[\mathrm{suc}\ x/x]}{\Gamma \vdash \mathrm{rec}_P(M, (x, p).M', \mathrm{zero}) \triangleright M : P[\mathrm{zero}/x]}$$

RED-RECS
$$\frac{\Gamma, x{:}\mathrm{Nat} \vdash P : U \quad \Gamma \vdash N : \mathrm{Nat} \quad \Gamma \vdash M : P[\mathrm{zero}/x] \quad \Gamma, x{:}\mathrm{Nat}, p{:}P \vdash M' : P[\mathrm{suc}\ x/x]}{\Gamma \vdash \mathrm{rec}_P(M, (x, p).M', \mathrm{suc}\ N) \triangleright M'[N/x, \mathrm{rec}_P(M, (x, p).M', N)/p] : P[\mathrm{suc}\ N/x]}$$

(a) Reduction

$\boxed{\Gamma \vdash M \equiv N : A}$ *(Equivalence)*

EQ-ETAPI
$$\frac{\Gamma \vdash M : \Pi x{:}A.\, B \quad \Gamma \vdash N : \Pi x{:}A.\, B \quad \Gamma, x{:}A \vdash M\ x \equiv N\ x : B}{\Gamma \vdash M \equiv N : \Pi x{:}A.\, B}$$

EQ-ETASIGMA
$$\frac{\Gamma \vdash M : \Sigma x{:}A.\, B}{\Gamma \vdash \langle \mathrm{fst}\ M, \mathrm{snd}\ M\rangle \equiv M : \Sigma x{:}A.\, B}$$

(b) Equivalence

Fig. 2. The remaining rules of CC_ω reduction and definitional equality

SECD

Instructions	I	::=	var x | st n | ty A | pushc ins | app | clo n ins | pair | fst | snd | unit | lit n | suc | rec P ins ins′
Instruction sequences	ins	::=	ret | I; ins
Values	v	::=	n | $\langle\rangle$ | A | $\langle$v, v′$\rangle$ | $\langle$env, ins$\rangle$
Runtime env./stack	env, st	::=	· | env :: v
Call frames	fr	::=	· | fr :: $\langle$ins, env, st$\rangle$
Machine configuration	C	::=	$\langle$ins, env, st, fr$\rangle$

Fig. 3. Syntax of the SECD machine

information on fr, loads the new runtime environment with env′ from the closure and the argument v, clears the stack, and loads the closure's code ins′ (SECD-RED-APP). The ret instruction marks the end of an instruction sequence. When ret is reached and the frame stack is non-empty, the machine discards the current stack except for the return value v on the top, restores the previous call frame, and places v on top of the restored stack (SECD-RED-RET).

SECD

Rule	Instruction	Env.	Stack	Frame
R	ins	env	st	fr
	ins′	env′	st′	fr′
	means that $\langle$ins, env, st, fr$\rangle \longrightarrow \langle$ins′, env′, st′, fr′$\rangle$ for rule SECD-RED-R.			
VAR	var x ; ins	env	st	fr
	ins	env	st :: env(x)	fr
ST	st n ; ins	env	st	fr
	ins	env	st :: st(n)	fr
TY	ty A ; ins	env	st	fr
	ins	env	st :: $A[\gamma]$	fr
	given a proof that env implements a closing substitution γ.			
PUSHC	pushc ins ; ins′	env	st	fr
	ins′	env	st :: $\langle$env, ins$\rangle$	fr
APP	app ; ins	env	st :: $\langle$env′, ins′$\rangle$ :: v	fr
	ins′	env′ :: v	·	fr :: $\langle$ins, env, st$\rangle$
CLO	clo n ins ; ins′	env	st :: $v_1, \ldots, v_n$	fr
	ins′	env	st :: $\langle v_1 :: \ldots :: v_n$, ins$\rangle$	fr
RECZ	rec P ins$_z$ ins$_s$; ins	env	st :: 0	fr
	ins$_z$	env	·	fr :: $\langle$ins, env, st$\rangle$
RECS	rec P ins$_z$ ins$_s$; ins	env	st :: $(n+1)$	fr
	rec P ins$_z$ ins$_s$; app ; ret	env	$\langle$env :: n, ins$_s\rangle$ :: n	fr :: $\langle$ins, env, st$\rangle$
RET	ret	env	st :: v	fr :: $\langle$ins′, env′, st′$\rangle$
	ins′	env′	st′ :: v	fr

Fig. 4. Operational semantics of dependent SECD

Evaluating recursor. The recursor instruction rec P ins$_z$ ins$_s$ recurses over the natural number n on top of the stack. When $n = 0$, the machine evaluates the zero case ins$_z$ (SECD-RED-RECZ). When $n = n' + 1$, the machine pushes a closure of the inductive case ins$_s$ with the current environment and the predecessor n' on the stack, and continues recursing over n'. The closure of the inductive case will eventually be applied to the result of recursing over n' (SECD-RED-RECS). Both cases are evaluated on a new call frame, which is not an efficient means of evaluation, but is necessary to prove type safety: the typing rule of allocating a new frame unifies the abstract value of the natural-number scrutinee from the type-checker's perspective and its concrete value at runtime, as we will see in rule SECD-FR-CONS.

2.3 Typing judgements for Instructions

We now assign dependent types to instructions of the SECD machine, using terms in CC$_\omega$ to model the stack's contents abstractly. The typing judgement of instructions (and sequences) has the form $\Gamma \vdash \text{ins} : \sigma \to \sigma'$, where σ is a list of well-typed CC$_\omega$ terms in context Γ, which we refer to as the *abstract stack*. The judgement can be read as follows: given an environment of type Γ, the instruction transforms the initial stack σ to σ'.

Formally, the typing judgement is only valid when all terms in abstract stacks σ and σ' are well-typed, i.e. $\Gamma \vdash M : A$ for every M in σ and σ'. We make this requirement implicit for simplicity of presentation. An abstract stack can equivalently be viewed as a substitution. Given a context Δ we write $\Gamma \vdash \sigma : \Delta$ to mean that σ is a well-typed substitution from Δ to Γ. Substitution is

well-typed in CC_ω, i.e. if $\Gamma \vdash \sigma : \Delta$ and $\Delta \vdash M : A$, then $\Gamma \vdash M[\sigma] : A[\sigma]$. The definitional equality extends pointwise to abstract stacks, denoted as $\Gamma \vdash \sigma \equiv \sigma'$.

The typing rules model the computation in the style of an abstract interpretation, keeping track of the contents on the stack. For example, $\mathtt{var}\ x$ copies the variable x from the context Γ to the stack σ (SECD-TY-VAR), and $\mathtt{ty}\ A$ pushes the CC_ω type A on the stack (SECD-TY-TY).

SECD

SECD-TY-VAR
$$\frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash \mathtt{var}\ x : \sigma \to \sigma :: x}$$

SECD-TY-TY
$$\frac{\Gamma \vdash A : U}{\Gamma \vdash \mathtt{ty}\ A : \sigma \to \sigma :: A}$$

Instruction $\mathsf{pushc}\ \mathsf{ins}$ pushes a function $\lambda x{:}A.\, M$ on the stack, where ins returns M in the current context extended with input $x : A$ (SECD-TY-PUSHC). Similarly, $\mathtt{clo}\ n\ \mathsf{ins}$ takes n terms from the stack as a well-typed substitution $\Gamma \vdash \delta : \Delta$ and pushes a function $(\lambda x{:}A.\, M)[\delta]$ on the stack (SECD-TY-CLO). Application is simple from the typing perspective: it transforms the stack from $\sigma :: M :: N$ to $\sigma :: M\ N$ (SECD-TY-APP). The return instruction requires the stack to be non-empty and takes the top item as the return value (SECD-TY-RET).

SECD

SECD-TY-PUSHC
$$\frac{\Gamma, x{:}A \vdash \mathsf{ins} : \cdot \to \sigma' :: M}{\Gamma \vdash \mathsf{pushc}\ \mathsf{ins} : \sigma \to \sigma :: \lambda x{:}A.\, M}$$

SECD-TY-CLO
$$\frac{\Gamma \vdash \delta : \Delta \qquad |\Delta| = n \qquad \Delta, x{:}A \vdash \mathsf{ins} : \cdot \to \sigma' :: M}{\Gamma \vdash \mathtt{clo}\ n\ \mathsf{ins} : \sigma :: \delta \to \sigma :: (\lambda x{:}A.\, M)[\delta]}$$

SECD-TY-APP
$$\frac{\Gamma \vdash M : \Pi x{:}A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{app} : \sigma :: M :: N \to \sigma :: M\ N}$$

SECD-TY-RET
$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathtt{ret} : \sigma :: M \to \sigma :: M}$$

Typing for the recursor instruction $\mathtt{rec}\ P\ \mathsf{ins}_z\ \mathsf{ins}_s$ appears complicated but is conceptually simple: given a predicate over natural numbers P, a base case ins_z that produces M, an inductive case ins_s that produces M' (in the current context extended with the predecessor x and the induction hypothesis p), the instruction recurses over the natural number term N on top of the stack and the result is $\mathrm{rec}_P(M, (x, p).M', N)$ (SECD-TY-REC).

SECD

SECD-TY-REC
$$\frac{\begin{array}{c}\Gamma, x{:}\mathrm{Nat} \vdash P : U \qquad \Gamma \vdash N : \mathrm{Nat} \\ \Gamma \vdash \mathsf{ins}_z : \cdot \to \sigma' :: M \qquad \Gamma \vdash M : P[\mathrm{zero}/x] \\ \Gamma, x{:}\mathrm{Nat}, p{:}P \vdash \mathsf{ins}_s : \cdot \to \sigma'' :: M' \qquad \Gamma, x{:}\mathrm{Nat}, p{:}P \vdash M' : P[\mathrm{suc}\ x/x]\end{array}}{\Gamma \vdash \mathtt{rec}\ P\ \mathsf{ins}_z\ \mathsf{ins}_s : \sigma :: N \to \sigma :: \mathrm{rec}_P(M, (x, p).M', N)}$$

Below shows the remaining typing rules, with straightforward rules for pairs, stack access, etc., and the conversion rule for equivalent abstract stacks (SECD-TY-EQUIV).

SECD

$$\textsc{secd-ty-Pair}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \mathsf{pair} : \sigma :: M :: N \to \sigma :: \langle M, N\rangle} \qquad \textsc{secd-ty-Fst}\ \frac{\Gamma \vdash M : \Sigma x{:}A.\,B}{\Gamma \vdash \mathsf{fst} : \sigma :: M \to \sigma :: \mathsf{fst}\ M}$$

$$\textsc{secd-ty-Snd}\ \frac{\Gamma \vdash M : \Sigma x{:}A.\,B}{\Gamma \vdash \mathsf{snd} : \sigma :: M \to \sigma :: \mathsf{snd}\ M} \qquad \textsc{secd-ty-Unit}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{unit} : \sigma \to \sigma :: \langle\rangle} \qquad \textsc{secd-ty-Lit}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{lit}\ n : \sigma \to \sigma :: \mathsf{n}}$$

$$\textsc{secd-ty-St}\ \frac{\vdash \Gamma \quad n < |\sigma| \quad \sigma(n) = M}{\Gamma \vdash \mathsf{st}\ n : \sigma \to \sigma :: M} \qquad \textsc{secd-ty-Seq}\ \frac{\Gamma \vdash \mathsf{l} : \sigma \to \sigma' \quad \Gamma \vdash \mathsf{ins} : \sigma' \to \sigma''}{\Gamma \vdash \mathsf{l}\ ;\ \mathsf{ins} : \sigma \to \sigma''} \qquad \textsc{secd-ty-Equiv}\ \frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' \quad \Gamma \vdash \sigma_2 \equiv \sigma_2' \quad \Gamma \vdash \mathsf{ins} : \sigma_1 \to \sigma_2}{\Gamma \vdash \mathsf{ins} : \sigma_1' \to \sigma_2'}$$

2.4 Machine Well-Formedness

Intuitively, a configuration $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle$ is well-formed if the instruction is well-typed and the other runtime components are compatible with the instruction's typing and with each other, so that the configuration can correctly step to the next state. We capture the notion of machine well-formedness with the following typing rules for values, environment, stack, and call frames:

(1) $\boxed{\mathsf{v} \vDash M : A}$ The runtime value v implements a closed CC_ω term M of type A.
(2) $\boxed{\mathsf{env} \vDash \gamma : \Gamma}$ The environment env implements a closing substitution $\cdot \vdash \gamma : \Gamma$ for context Γ.
(3) $\boxed{(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma}$ The stack st implements the abstract stack σ, given some env that implements $\cdot \vdash \gamma : \Gamma$.
(4) $\boxed{(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{fr} : [M : A]}$ The stack of call frames fr is well-formed, and its top frame expects its callee to return a value compatible to M (of type A) under the top frame's environment $(\mathsf{env} :_\gamma \Gamma)$.

We use these typing judgements to relate the open, abstract terms at compile time to the closed, concrete terms at runtime. These judgements also admit conversion rules for definitionally equivalent CC_ω terms (similar to SECD-TY-EQUIV), but we omit them here for brevity.

Environment and values. The judgements for runtime values $\mathsf{v} \vDash M : A$ and for environments $\mathsf{env} \vDash \gamma : \Gamma$ are defined mutual-inductively. Natural numbers, the unit value, and closed types implement themselves. A closure $\langle \mathsf{env}, \mathsf{ins}\rangle$ implements $(\lambda x{:}A.\,M)[\gamma]$ if the instruction produces $\Gamma, x{:}A \vdash M : B$ and the environment implements Γ as γ (SECD-VAL-CLO). If v implements M and v' implements N, then the pair $\langle \mathsf{v}, \mathsf{v}'\rangle$ implements $\langle M, N\rangle$ (SECD-VAL-PAIR). An empty environment implements the empty substitution ϵ. If $\mathsf{env} \vDash \gamma : \Gamma$ and v implements a closed term M of type $A[\gamma]$, then $\mathsf{env} :: \mathsf{v}$ implements the extended context $\Gamma, x : A$ with an extended substitution $\gamma, M/x$ (SECD-ENV-CONS).

Runtime stack. The judgement for runtime stacks $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma$ states that, given some env that implements Γ as γ, st implements the abstract stack σ. We check if $(\mathsf{env} :_\gamma \Gamma)$ is well-typed at the base case. Given st that implements σ, $\mathsf{st} :: \mathsf{v}$ implements $\sigma :: M$ (where $\Gamma \vdash M : A$) if the value v implements some closed term M' that is equivalent to $M[\gamma]$, i.e. the term M closed by the environment (SECD-ST-CONS).

SECD

$$\textsc{secd-val-Clo}\ \frac{\Gamma, x{:}A \vdash M : B \quad \mathsf{env} \vDash \gamma : \Gamma \quad \Gamma, x{:}A \vdash \mathsf{ins} : \cdot \longrightarrow \sigma :: M}{\langle \mathsf{env}, \mathsf{ins} \rangle \vDash (\lambda x{:}A.\, M)[\gamma] : (\Pi x{:}A.\, B)[\gamma]}$$

$$\textsc{secd-val-Pair}\ \frac{x{:}A \vdash B : U \quad \mathsf{v} \vDash M : A \quad \mathsf{v}' \vDash N : B[M/x]}{\langle \mathsf{v}, \mathsf{v}' \rangle \vDash \langle M, N \rangle : \Sigma x{:}A.\, B}$$

$$\textsc{secd-env-Cons}\ \frac{\Gamma \vdash A : U \quad \mathsf{env} \vDash \gamma : \Gamma \quad \mathsf{v} \vDash M : A[\gamma]}{\mathsf{env} :: \mathsf{v} \vDash \gamma, M/x : \Gamma, x{:}A}$$

$$\textsc{secd-st-Cons}\ \frac{(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma \quad \mathsf{v} \vDash M' : A[\gamma] \quad \Gamma \vdash M : A \quad \vdash M[\gamma] \equiv M' : A[\gamma]}{(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} :: \mathsf{v} \vDash \sigma :: M}$$

Call frames. Having a well-typed runtime environment and stack ensures that the instructions can correctly manipulate values within the current call frame. Things become more complex when we need to transport values between different frames upon function calls and returns.

Recall that in rule SECD-RED-RET, the return value v is moved directly from the callee frame with environment $(\mathsf{env} :_\gamma \Gamma)$ to the caller frame with environment $(\mathsf{env}' :_{\gamma'} \Gamma')$. From the perspective of instruction typing, we are treating two different CC_ω terms as the same: the caller's expected result $\Gamma' \vdash M' : A'$, and callee's produced result $\Gamma \vdash M : A$. We must ensure that the concrete value v implements both M and M' with respect to their corresponding runtime environments, i.e. establish that $\mathsf{v} \vDash N$ for some $N \equiv M[\gamma] \equiv M'[\gamma']$.

SECD

$$\textsc{secd-fr-Cons}\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{ins} : \sigma :: M \longrightarrow \sigma' :: N \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B \\ \mathsf{env} \vDash \gamma : \Gamma \quad (\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma \quad (\mathsf{env}' :_{\gamma'} \Gamma') \vdash \mathsf{fr} : [N' : B'] \\ \vdash B[\gamma] \equiv B'[\gamma'] : U \quad \vdash N[\gamma] \equiv N'[\gamma'] : B[\gamma]\end{array}}{(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{fr} :: \langle \mathsf{ins}, \mathsf{env}, \mathsf{st} \rangle : [M : A]}$$

The typing judgement $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{fr} : [M : A]$ is defined exactly for this purpose. It states that the top frame of fr has $(\mathsf{env} :_\gamma \Gamma)$ as the environment and it expects a term $\Gamma \vdash M : A$ as return. The empty frame stack accepts any return value. To push a new frame $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st} \rangle$ on fr, the following conditions are necessary: ins is well-typed and it will produce a term $\Gamma \vdash N : B$ for its caller; the runtime environment and stack are well-typed and compatible with ins; fr is well-formed and its top frame has environment $(\mathsf{env}' :_{\gamma'} \Gamma')$ and expects a return value $\Gamma' \vdash N' : B'$; the two return values and their types are compatible with respect to their own environments, i.e. $B[\gamma] \equiv B'[\gamma']$ and $N[\gamma] \equiv N'[\gamma']$. Finally, the judgement records the new frame's environment $(\mathsf{env} :_\gamma \Gamma)$ and its expected return $[M : A]$ (SECD-FR-CONS).

Configuration well-formedness. Informally, a configuration is well-formed if the instruction sequence is well-typed, and the runtime environment, stack, and call frames are compatible with the instructions and with each other. In particular, conditions (5) and (6) state that the current frame's instruction sequence should return a term that is compatible with the expected return of the previous call frame, as we discussed in SECD-FR-CONS.

DEFINITION 2.6 (CONFIGURATION WELL-FORMEDNESS). *A configuration* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$ *is well-formed, denoted as* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$*, if the following conditions hold:*

(1) $\Gamma \vdash \mathsf{ins} : \sigma \longrightarrow \sigma' :: M$ *for some* Γ, σ, σ', *and* M, *with* $\Gamma \vdash M : A$ *for some* A.
(2) $\mathsf{env} \vDash \gamma : \Gamma$ *for some* γ.
(3) $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma$.
(4) $(\mathsf{env}' :_{\gamma'} \Gamma') \vdash \mathsf{fr} : [M' : A']$, *for some* env', γ', Γ', M', *and* A'.

(5) $\vdash A[\gamma] \equiv A'[\gamma'] : U$.
(6) $\vdash M[\gamma] \equiv M'[\gamma'] : A[\gamma]$.

Structural properties of the judgements. We finally state two important structural properties of the typing judgements of dependent SECD machine. First, the parameters of the judgements are well-formed (Lemma 2.7). Second, well-typed runtime values have canonical forms (Lemma 2.8). For example, a value of type Nat must be a natural number, a value of type $\Pi x{:}A.\, B$ must be a closure, etc. Proving the canonicity of values requires non-confusion of type formers (Lemma 2.3) in the specification language, for example, if $\mathsf{v} \vDash M : \Pi x{:}A.\, B$, then the judgement must be derived from SECD-VAL-CLO since Π-types do not confuse with other types, therefore, v must be a closure. We write $A \equiv_{nf} A'$ for A' being the normal form of the closed type A as a convention.

Lemma 2.7 (Well-formedness of typing judgements). *The parameters of the judgements are well-formed in the following sense:*

- *If* $\Gamma \vdash \mathsf{ins} : \sigma \longrightarrow \sigma'$, *then* $\vdash \Gamma$ *and every term in* (σ, σ') *is well-typed under* Γ.
- *If* $\mathsf{env} \vDash \gamma : \Gamma$, *then* $\cdot \vdash \gamma : \Gamma$ *and* $|\mathsf{env}| = |\Gamma|$.
- *If* $\mathsf{v} \vDash M : A$, *then* $\vdash M : A$.
- *If* $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma$, *then* $\mathsf{env} \vDash \gamma : \Gamma$, $|\mathsf{st}| = |\sigma|$, *and every term in* σ *is well-typed under* Γ.
- *If* $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{fr} : [M : A]$, *then* $\mathsf{env} \vDash \gamma : \Gamma$ *and* $\Gamma \vdash M : A$.

Lemma 2.8 (Canonicity of values). *If* $\mathsf{v} \vDash M : A$, *then*

- *If* $A \equiv_{nf} U_i$, *then* $\mathsf{v} = A'$ *(*v *is a type).*
- *If* $A \equiv_{nf} Nat$, *then* $\mathsf{v} = n$ *for some natural number* n.
- *If* $A \equiv_{nf} Unit$, *then* $\mathsf{v} = \langle\rangle$.
- *If* $A \equiv_{nf} \Pi x{:}A'.\, B'$, *then* $\mathsf{v} = \langle \mathsf{env}, \mathsf{ins} \rangle$ *for some* $\mathsf{env}, \mathsf{ins}$.
- *If* $A \equiv_{nf} \Sigma x{:}A'.\, B'$, *then* $\mathsf{v} = \langle \mathsf{v}_1, \mathsf{v}_2 \rangle$ *for some* $\mathsf{v}_1, \mathsf{v}_2$.

2.5 Type Safety and Partial Correctness

The dependent SECD machine has type safety by progress and preservation: a well-formed machine either halts or transitions to the next state, and well-formed machines transition to well-formed next states. A machine configuration halts if it eventually evaluates to a final state where the instruction is `ret`, the stack of call frames is empty, and there is a value on top of the stack as the final result.

Definition 2.9 (Halt). $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$ *halts with result* v, *denoted as* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \Downarrow \mathsf{v}$, *if* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \longrightarrow^* \langle \mathtt{ret}, \mathsf{env}', \mathsf{st}' :: \mathsf{v}, \cdot \rangle$.
$\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$ *halts, denoted as* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \Downarrow$, *if there exists* v *such that* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \Downarrow \mathsf{v}$.

We prove type safety by induction on the instruction sequence ins, and the proof requires canonicity of values, injectivity of type formers, and structural properties of the runtime environments and stacks (e.g. values found on well-formed environments are well-typed).

Theorem 2.10 (Progress). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$, *then either* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \longrightarrow \langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}' \rangle$ *or* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$ *halts.*

Theorem 2.11 (Preservation). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$ *and* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \longrightarrow \langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}' \rangle$, *then* $\vdash \langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}' \rangle$.

We obtain type safety from progress and preservation: a well-formed dependent SECD configuration never evaluates to an illegal state; it either halts or keeps evaluating forever.

Corollary 2.12 (Type safety). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle$, *then it never reduces to a configuration that is irreducible and not at halt.*

Let us examine a well-formed configuration $\langle \mathtt{ret}, \mathsf{env}, \mathsf{st} :: \mathsf{v}, \cdot\rangle$ at halt. The type of the return is $\Gamma \vdash \mathtt{ret} : \sigma :: M \longrightarrow \sigma :: M$, the environment implements some closing substitution γ, the stack of call frames is empty, and the stack has a return value v at the top. Our typing scheme gives an extremely precise description for the return value v: it must be a value that implements $M[\gamma]$, the abstract final result specified by the typing judgement concretized by the runtime environment. By type safety, if a well-formed configuration halts, the halting state is also well-formed. It implies that the dependent SECD machine has a *partial correctness* property: if an instruction sequence is well-typed and executing it with compatible environment and stack halts, then it always produces the correct value according to its type specification.

Corollary 2.13 (Partial correctness). *If* $\Gamma \vdash \mathsf{ins} : \sigma \longrightarrow \sigma' :: M$ *with* $\Gamma \vdash M : A$, $\mathsf{env} \vDash \gamma : \Gamma$, $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma$, *and* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \cdot\rangle \Downarrow \mathsf{v}$, *then* $\mathsf{v} \vDash M[\gamma] : A[\gamma]$.

2.6 Termination and Total Correctness

In this section, we prove the final key property of the SECD machine: well-formed machines always halt. Combining partial correctness with termination yields *total correctness*: a well-formed machine always terminates and produces the value specified by its type. We first define a logical relation $H(A)$ for halting runtime values of type A (Definition 2.15) and show that a well-formed configuration always terminates at its current frame (Lemma 2.21) by proving a fundamental lemma for the logical relation (Lemma 2.19). Termination then follows from an induction over the size of the call frame stack (Theorem 2.22).

Definition 2.14 (Halt at current frame). $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle$ *halts at its current frame with return value* v, *denoted as* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle \downarrow \mathsf{v}$, *if* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle \longrightarrow^* \langle \mathtt{ret}, \mathsf{env}, \mathsf{st}' :: \mathsf{v}, \mathsf{fr}\rangle$ *and the reduction sequence never returns beyond* fr.
$\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle$ *halts at its current frame, denoted as* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle \downarrow$, *if there exists* v *such that* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle \downarrow \mathsf{v}$.

The halting relation $H(A)$ is a family of subsets of SECD machine values indexed over closed types of CC_ω. We define $H(A)$ recursively over the derivation of $\Vdash A$, the judgement for A being a closed type with a normal form, inspired by Abel et al. [2018]. The normal form of A must be one of the following: the natural number type, the unit type, a universe, a Π-type, or a Σ-type. For the first three cases, $\Vdash A$ holds without further requirement. If the normal form of A is $\Pi x{:}A'.B'$, then $\Vdash A$ holds if $\Vdash A'$ and $\Vdash B'[a/x]$ for all $a : A'$, and similarly for the case of Σ-type. Since CC_ω is normalizing, we can derive $\Vdash A$ for every well-typed closed type A.

CC_ω

$$\textsc{nf-Nat}\ \frac{A \equiv_{nf} \mathrm{Nat}}{\Vdash A} \qquad \textsc{nf-Pi}\ \frac{A \equiv_{nf} \Pi x{:}A'.B' \quad \Vdash A' \quad \forall (a : A').\ \Vdash B'[a/x]}{\Vdash A} \qquad \textsc{nf-Sigma}\ \frac{A \equiv_{nf} \Sigma x{:}A'.B' \quad \Vdash A' \quad \forall (a : A').\ \Vdash B'[a/x]}{\Vdash A}$$

We can now define $H(A)$ by recursion on the derivation tree of $\Vdash A$. The definition is straightforward when the normal form of A is natural number, the unit type, or a universe.

In the case of $A \equiv_{nf} \Sigma x{:}A'.B'$ (i.e. $\Vdash A$ by nf-Sigma), $\mathsf{v} \in H(A)$ if v is a well-typed pair $\langle \mathsf{v}_1, \mathsf{v}_2\rangle$ and its two components are in the relations of the first/second projection. Specifically, if v implements some closed term M, then we need $\mathsf{v}_1 \in H(A')$ and $\mathsf{v}_2 \in H(B'[\mathrm{fst}\ M/x])$ (since B' depends on the

first projection of M). Note that $H(B'[\text{fst } M/x])$ is well-defined by recursion on the sub-derivation $\forall\,(a : A').\ \Vdash B'[a/x]$ instantiated with $a = \text{fst } M$.

When $A \equiv_{nf} \Pi x{:}A'.\,B'$, $\mathsf{v} \in H(A)$ if v is a well-typed closure $\langle \mathsf{env}, \mathsf{ins}\rangle$, and for all $\mathsf{v}' \in H(A')$ with $\mathsf{v}' \vDash a : A'$, evaluating the closure's code in $\mathsf{env} :: \mathsf{v}'$ with an empty stack halts at the current frame and the return value is in $H(B'[a/x])$. This definition uses the common 'logical' condition for function types: for all values in the relation of the input type, applying the closure to the value produces a result in the relation of the output type.

Definition 2.15 (The halting relation on values). *Let $H(A)$ be a subset of SECD-machine values indexed over closed types of CC_ω. We define $H(A)$ recursively on the derivation of $\Vdash A$.*

- NF-Nat: $A \equiv_{nf} Nat$ $\mathsf{v} \in H(A)$ *iff* $\mathsf{v} = n$.
- NF-Unit: $A \equiv_{nf} Unit$ $\mathsf{v} \in H(A)$ *iff* $\mathsf{v} = \langle\rangle$.
- NF-Universe: $A \equiv_{nf} U_i$ $\mathsf{v} \in H(A)$ *iff* $\mathsf{v} \vDash A' : U_i$.
- NF-Pi: $A \equiv_{nf} \Pi x{:}A'.\,B'$ $\mathsf{v} \in H(A)$ *iff* $\mathsf{v} = \langle \mathsf{env}, \mathsf{ins}\rangle$, $\mathsf{v} \vDash M : \Pi x{:}A'.\,B'$, *and for all* $\mathsf{v}' \in H(A')$ *such that* $\mathsf{v}' \vDash a : A'$, $\langle \mathsf{ins}, \mathsf{env} :: \mathsf{v}', \cdot, \mathsf{fr}\rangle \downarrow \mathsf{v}''$ *and* $\mathsf{v}'' \in H(B'[a/x])$.
- NF-Sigma: $A \equiv_{nf} \Sigma x{:}A'.\,B'$ $\mathsf{v} \in H(A)$ *iff* $\mathsf{v} = \langle \mathsf{v}_1, \mathsf{v}_2\rangle$, $\mathsf{v} \vDash M : \Sigma x{:}A'.\,B'$, $\mathsf{v}_1 \in H(A')$, $\mathsf{v}_2 \in H(B'[\mathit{fst}\ M/x])$.

Lemma 2.16 (Well-typed halting values). *If $\mathsf{v} \in H(A)$, then $\mathsf{v} \vDash M : A$ for some M.*

The halting relations for well-typed environments $\mathsf{env} \Vdash_h \gamma : \Gamma$ and stacks $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \Vdash_h \sigma$ are given in judgemental form, defined recursively on their type derivations. Both judgements mean that every value in the environment/stack is in the halting relation of its corresponding type.

Definition 2.17 (The halting relation on environments). *Let $\mathsf{env} \Vdash_h \gamma : \Gamma$ be a subset of well-typed runtime environments $\mathsf{env} \vDash \gamma : \Gamma$, defined on the typing derivation of $\mathsf{env} \vDash \gamma : \Gamma$ recursively:*

- SECD-env-Nil $\cdot \Vdash_h \epsilon : \cdot$ *always hold.*
- SECD-env-Cons $(\mathsf{env} :: \mathsf{v}) \Vdash_h (\gamma, M/x) : (\Gamma, x{:}A)$ *if* $\mathsf{env} \Vdash_h \gamma : \Gamma$ *and* $\mathsf{v} \in H(A[\gamma])$.

Definition 2.18 (The halting relation on stacks). *Let $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \Vdash_h \sigma$ be a subset of well-typed runtime stacks $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma$, defined on the stack typing derivation of $\mathsf{env} \vDash \gamma : \Gamma$ recursively:*

- SECD-st-Nil $(\mathsf{env} :_\gamma \Gamma) \vdash \cdot \Vdash_h \cdot$ *holds if* $\mathsf{env} \vDash \gamma : \Gamma$.
- SECD-st-Cons $(\mathsf{env} :_\gamma \Gamma) \vdash (\mathsf{st} :: \mathsf{v}) \Vdash_h (\sigma :: M)$ *if* $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \Vdash_h \sigma$ *and* $\mathsf{v} \in H(A[\gamma])$ *(where $\Gamma \vdash M : A$).*

We are now ready to state and prove the fundamental lemma: given a well-typed instruction ins and compatible environment env and stack st in halt, then for every frame fr, the configuration $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle$ halts at the current frame with a halting value as the result. Note that halting at the current frame means that the stack of call frames never goes beyond fr, so we do not require fr to be well-formed, and preservation of machine well-formedness still holds for reductions that do not go beyond fr.

Lemma 2.19 (Fundamental lemma). *For all $(\mathsf{ins}, \mathsf{env}, \mathsf{st})$, if $\Gamma \vdash \mathsf{ins} : \sigma \to \sigma' :: M$ with $\Gamma \vdash M : A$, $\mathsf{env} \Vdash_h \gamma : \Gamma$, and $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \Vdash_h \sigma$, then $\forall \mathsf{fr}.\langle \mathsf{ins}, \mathsf{st}, \mathsf{fr}\rangle \downarrow \mathsf{v}$ and $\mathsf{v} \in H(A[\gamma])$.*

The fundamental lemma is proved by an induction on ins. From the fundamental lemma, we can immediately show that well-typed values, environments, and stacks are in their respective

halting relations by induction, where the fundamental lemma is used in the closure case of values (Lemma 2.20). Then, given $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$, since the environment and the stack are well-typed, they are in the halting relation and we have $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \downarrow \mathsf{v}$ (Lemma 2.21). In other words, all well-formed machines halt at the current frame with the result in the halting relation. We finally prove that well-formed machines eventually halt by an induction on the size of the call frame stack (Theorem 2.22).

Lemma 2.20 (Well-typed values halt). *Well-typed values, environments, and stacks are in their respective halting relations.*

- *If* $\mathsf{v} \vDash a : A$, *then* $\mathsf{v} \in H(A)$.
- *If* $\mathsf{env} \vDash \gamma : \Gamma$, *then* $\mathsf{env} \Vdash_h \gamma : \Gamma$.
- *If* $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma$, *then* $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \Vdash_h \sigma$.

Lemma 2.21 (Termination at current frame). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$, *then* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \downarrow \mathsf{v}$ *and* $\mathsf{v} \in H(A)$ *where* $\mathsf{v} \vDash M : A$.

Theorem 2.22 (Termination). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle$, *then* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle \Downarrow \mathsf{v}$ *for some* v.

We can now combine termination with partial correctness to obtain total correctness: a well-formed machine always terminates and produces the value specified by its type (Corollary 2.23). The corresponding result for complete programs (i.e. closed instruction sequences with empty initial stacks) follows as a corollary (Corollary 2.24).

Corollary 2.23 (Total correctness). *If* $\Gamma \vdash \mathsf{ins} : \sigma \longrightarrow \sigma' :: M$ *with* $\Gamma \vdash M : A$, $\mathsf{env} \vDash \gamma : \Gamma$, *and* $(\mathsf{env} :_\gamma \Gamma) \vdash \mathsf{st} \vDash \sigma$, *then* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \cdot \rangle \Downarrow \mathsf{v}$ *with* $\mathsf{v} \vDash M[\gamma] : A[\gamma]$.

Corollary 2.24 (Total correctness for programs). *If* $\cdot \vdash \mathsf{ins} : \cdot \longrightarrow \sigma :: M$ *with* $\vdash M : A$, *then* $\langle \mathsf{ins}, \cdot, \cdot, \cdot \rangle \Downarrow \mathsf{v}$ *with* $\mathsf{v} \vDash M : A$.

2.7 Compilation

In this section, we define a compilation function from CC_ω to the dependent SECD machine to demonstrate that our typing scheme for dependently typed abstract machines fulfils our claims in the introduction: type-checking the compiled code guarantees correctness of whole-program compilation and separate compilation.

The compilation function $[\![-]\!]_c$ is defined in Fig. 5a. It recursively maps every term in CC_ω to its corresponding instruction in dependent SECD. We also define a mapping $[\![-]\!]_v$ in Fig. 5b that transforms closed CC_ω values to SECD values. The compilation scheme is shown type-preserving by a simple induction (Lemma 2.25). Therefore, by total correctness, a closed CC_ω term M compiles to an SECD program that terminates and produces a result implementing M (Corollary 2.26). Similarly, given an open CC_ω term M and a list of closed CC_ω values γ, executing $[\![M]\!]_c$ with the compiled values in the environment terminates and produces a value implementing $M[\gamma]$, meaning that we can compile open terms separately and link them together at runtime (Corollary 2.27).

Note that the correctness theorems of the compilation function do not rely on how it is defined; they only require the compiler to preserve the types. It is possible to define different compilation schemes (for example, one that involves common optimizations such as function inlining or constant propagation) and still enjoy these correctness properties as long as types are preserved.

Lemma 2.25 (Type preservation). *If* $\Gamma \vdash M : A$, *then* $\Gamma \vdash [\![M]\!]_c : \cdot \longrightarrow \cdot :: M$. *If* $\vdash M : A$ *and* M *is a value, then* $[\![M]\!]_v \vDash M : A$.

Corollary 2.26 (Correctness of whole-program compilation). *If* $\cdot \vdash M : A$, *then* $\langle [\![M]\!]_c, \cdot, \cdot, \cdot \rangle \Downarrow \mathsf{v}$ *with* $\mathsf{v} \vDash M : A$.

$[\![\mathrm{CC}_\omega]\!]_c$

$$[\![x]\!] = \mathtt{var}\ x \qquad [\![\lambda x{:}A.\,M]\!] = \mathtt{pushc}\ [\![M]\!]_c \qquad [\![M\ N]\!] = [\![M]\!]; [\![N]\!]; \mathtt{app}$$

$$[\![\Pi x{:}A.\,B]\!] = \mathtt{ty}\ \Pi x{:}A.\,B \qquad [\![\langle M, N\rangle]\!] = [\![M]\!]; [\![N]\!]; \mathtt{pair} \qquad [\![\mathrm{fst}\ M]\!] = [\![M]\!]; \mathtt{fst}$$

$$[\![\mathrm{rec}_P(M, (x,p).M', N)]\!] = [\![N]\!]; \mathtt{rec}\ P\ [\![M]\!]_c\ [\![M']\!]_c \qquad [\![M]\!]_c = [\![M]\!]; \mathtt{ret}$$

(a) Term compilation

$$[\![\lambda x{:}A.\,M]\!]_v = \langle \cdot, [\![M]\!]_c \rangle \quad [\![\langle M, N\rangle]\!]_v = \langle [\![M]\!]_v, [\![N]\!]_v \rangle \quad [\![n]\!]_v = n \quad [\![A]\!]_v = A \quad [\![\langle\rangle]\!]_v = \langle\rangle$$

(b) Value compilation

Fig. 5. Compilation from CC_ω to SECD

Corollary 2.27 (Correctness of separate compilation). *If* $\Gamma \vdash M : A$ *and have closed* CC_ω *values* $\gamma = M_1, ..., M_n$ *such that* $\cdot \vdash \gamma : \Gamma$, *then* $\langle [\![M]\!]_c, [\![M_1]\!]_v :: ... :: [\![M_n]\!]_v, \cdot, \cdot \rangle \Downarrow \mathtt{v}$ *with* $\mathtt{v} \vDash M[\gamma] : A[\gamma]$.

2.8 Summary

We developed the dependent SECD machine as a theoretical foundation for dependently typed assembly, with type safety, termination, and support for safe separate compilation for type-preserving compilers. Some aspects of its design do not correspond to an efficient low-level implementation, such as storing instruction sequences on the stack (inside closures), keeping a proof of environment well-formedness for executing `ty` A, and nested instructions like `pushc`. These features should not cause practical concerns, as they could be eliminated by transforming the source language CC_ω to a dependently typed intermediate representation as the specification language and then deriving instructions of a lower-level abstract machine. The next section applies the principles of §2 to the *Defunctionalized* Calculus of Constructions [Huang and Yallop 2023a] and derives an abstract machine that is closer to typed assembly languages [Morrisett et al. 2002, 1998].

3 Dependent Abstract Machine

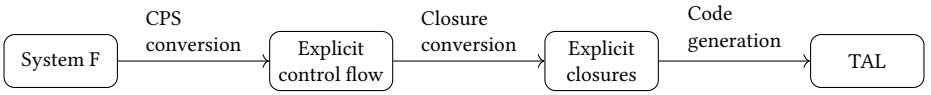
In this section, we present the dependent abstract machine (DAM) that is a suitable target language for type-preserving compilation with dependent types, and sketch a compilation scheme from CC_ω to it. The syntax of DAM programs is in the style of typed assembly languages [Morrisett et al. 2002, 1998]. Its operational model is a variation of the SECD machine we saw in §2.2, and the specification language is based on the Defunctionalized Calculus of Constructions (DCC) [Huang and Yallop 2023a], an intermediate representation for defunctionalization with dependent types.

§3.1 first recalls Morrisett et al. [1998]'s TAL and gives an overview of DAM and its compilation scheme. §§3.2 and 3.3 then present the specification language DCC and the syntax of DAM. §3.4 gives typing rules for its instructions and machine configurations that guarantee type safety. §3.5 shows that well-typed configurations terminate by a translation back to the dependent SECD machine, from which we obtain total correctness for DAM. Finally, §3.6 defines a type-preserving compilation function from DCC to DAM that supports safe separate compilation (Corollary 3.19).

3.1 Overview: Typed Assembly Languages

We recall type-preserving compilation and typed assembly languages *without* fully dependent types, then give an overview of our compilation scheme for CC_ω. We start by summarizing Morrisett

et al. [1998]'s influential typed assembly language TAL, whose compilation scheme is depicted below:



TAL is based on a RISC-style instruction set for an abstract register machine. Starting from the source language System F, the compiler applies a series of type-preserving transformations targeting typed intermediate languages and eventually TAL, removing high-level language features and preserving the types at each stage. The compiler begins with the *continuation-passing style* (CPS) transformation [Harper and Lillibridge 1993] to make control flow explicit. It then performs *closure conversion* [Minamide et al. 1996] with *hoisting*, identifying all function closures and lifting them to the top level. The final stage is code generation that includes explicit allocation for tuples and register allocation for variables.

A TAL program consists of a list of code blocks, each corresponding to a hoisted closure in a compiled program, and a `main` block. A code block starts with a type annotation, specifying the expected types of the values stored in registers when control enters the block. For example, the TAL program below computes 5! and puts the final result in register `r1`. The blocks `fact` and `nonzero` each specify that registers `r1` and `r2` should both contain an integer before the block is executed.

TAL

```
1 main: code[]{.                          8 nonzero:
2   mov r2, 5                                 code[]{r1 : int, r2 : int}.
3   mov r1, 1                             9   mul r1, r1, r2
4   jmp fact                             10   sub r2, r2, r1
5 fact: code[]{r1: int, r2 : int}.       11   jmp fact
6   bnz r2, nonzero
7   halt[int]
```

TAL's type checker ensures that registers have the correct types before jumping to a code block and rules out ill-typed operations to achieve type safety. Variants of TAL include TALx86 [Crary et al. 1999] that targets Intel processors and STAL [Morrisett et al. 2002] that is based on a stack machine and does not need CPS conversion for compilation.

Previous work also used the *singleton* type in attempts to bring dependent types down to the assembly level, relating compile-time and runtime values with $Single_A(v)$, whose only inhabitant is v of type A. DTAL [Xi and Harper 2001] used integer-valued singletons $Single_{int}(x)$ to specify registers containing the integer x, and the singleton type is the definitive feature in Winwood and Chakravarty [2011]'s Singleton language. However, singletons in these systems are cumbersome to use for specifying computations, which often requires extra proof terms, and are difficult to generate from source programs that do not use singletons throughout.

Our dependent abstract machine (DAM) instead uses terms in a specification calculus to model the stack's context and uses typing rules to relate runtime and compile-time values as discussed in §2. Our compilation scheme from CC_ω to DAM follows TAL's approach, and is depicted below:



We compile to DAM with dependently typed defunctionalization (similar to closure conversion with hoisting) through DCC and code generation. Similar to STAL, we compile without CPS transformation. Like TAL, a DAM program consists of a list of code blocks and a main block. Each

block starts with a type annotation that specifies the code's computational behaviour with DCC terms, and the instruction set is nearly identical to the instructions of the SECD machine. The DAM program below computes the factorial of 5. The code blocks of mult (Line 1, definition omitted) and fact (Lines 12–15) are defunctionalized labels generated from functions in the source code. The base and inductive cases of recursors in the source program similarly become code blocks ℓ_z (Lines 2–3) and ℓ_s (Lines 5–11).

DAM

```
1   mult   ({x:Nat}, y:Nat ↦ x ∗ y : Nat) :
2     ℓz   ({y:Nat}, _:Unit ↦ 1 : Nat) :
3          lit 1
4          ret
5     ℓs   ({y : Nat, x : Nat}, p : Nat ↦
           (sucx) ∗ p : Nat) :
6          var x
7          suc
8          clo 1 mult
9          var p
10         app
11         ret
12  fact   ({}, y : Nat ↦
           rec_Nat(1, (x, p).(suc x) ∗ p, y) : Nat) :
13         var y
14         rec P ℓz ℓs
15         ret
16  main   (· ⊢ 120 : Nat) :
17         clo 0 fact
18         lit 5
19         app
20         ret
```

The main block (Lines 16–20) starts with an empty stack and environment. The first instruction clo (Line 17) forms closure fact{} on the stack. Next (Line 18), `lit` 5 pushes value 5 on the stack and app applies the closure to the argument on top of the stack (Line 19), producing the final result $\mathsf{rec}_{\mathsf{Nat}}(1, (x, p).(\mathsf{suc}\ x) * p, 5)$, which reduces to 120 in DCC, matching the specification on Line 16. In contrast to the factorial in TAL, which only promises to return a *well-typed* result (if it halts), DAM promises to give the correct return *value* by keeping track of the stack contents, as a result of total correctness (Corollary 3.12).

3.2 Specification Language: Defunctionalized Calculus of Constructions

We now present the defunctionalized calculus of constructions (DCC) [Huang and Yallop 2023a] extended with dependent sums, unit type and natural numbers, with large elimination and typed equivalence. The syntax of DCC is given in Fig. 6. We use a sans-serif font to distinguish DCC terms from CC_ω terms.

DCC

Expressions	A, B, M, N, P	::=	$\mathsf{x} \mid \mathsf{U}_i \mid \Pi \mathsf{x}{:}\mathsf{A}.\,\mathsf{B} \mid \mathsf{M}\ \mathsf{N} \mid \ell\{\sigma\}$
			$\mid \Sigma \mathsf{x}{:}\mathsf{A}.\,\mathsf{B} \mid \langle \mathsf{M}, \mathsf{N} \rangle \mid \mathsf{fst}\ \mathsf{M} \mid \mathsf{snd}\ \mathsf{M}$
			$\mid \mathsf{Unit} \mid \langle\rangle \mid \mathsf{Nat} \mid \mathsf{zero} \mid \mathsf{suc}\ \mathsf{M} \mid \mathsf{rec}_{\mathsf{P}}(\mathsf{M}, (\mathsf{x}, \mathsf{p}).\mathsf{M}', \mathsf{N})$
Type contexts	Γ, Δ	::=	$\cdot \mid \Gamma, \mathsf{x}{:}\mathsf{A}$
Label contexts	$\mathfrak{D}$	::=	$\cdot \mid \mathfrak{D}, \ell(\{\Delta\}, \mathsf{x}{:}\mathsf{A} \mapsto \mathsf{M} : \mathsf{B})$

Fig. 6. Syntax of DCC

Defunctionalized labels. The expressions of DCC are almost identical to CC_ω, with one notable difference: DCC contains first-class function labels (accessible from a label context $\mathfrak{D}$) instead of lambda abstractions. The only way to create a function is forming a closure $\ell\{\sigma\}$ by instantiating a label $\ell \in \mathfrak{D}$ with a list of terms $\sigma = \mathsf{M}_1, \ldots, \mathsf{M}_n$ that provides values for its free variables.

An entry of the label context $\ell(\{\Delta\}, x{:}A \mapsto M : B)$ records how ℓ should be type-checked and evaluated: Δ specifies the type of the environment that ℓ expects; $(x : A)$ and B are the input and return types of the closure; M is the label's body, i.e. the expression to which the formed closure reduces upon application.

The typing judgement of DCC has the form $\mathfrak{D}; \Gamma \vdash M : A$. If a label ℓ expects an environment of type Δ and the list of terms σ forms a well-typed substitution from Δ to Γ, then the closure $\ell\{\sigma\}$ is a well-typed function $\Pi x{:}A[\sigma].\, B[\sigma]$ (DCC-TY-LABEL). An application $\ell\{\sigma\}\ N$ is β-reduced to the label's body M with σ substituted for its free variables and N for the input x (DCC-RED-BETA).

DCC

$$\textsc{dcc-ty-Label}\ \frac{\mathfrak{D}; \Gamma \vdash \sigma : \Delta \qquad \ell(\{\Delta\}, x{:}A \mapsto M : B) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \ell\{\sigma\} : \Pi x{:}A[\sigma].\, B[\sigma]}$$

$$\textsc{dcc-red-Beta}\ \frac{\ell(\{\Delta\}, x{:}A \mapsto M : B) \in \mathfrak{D} \qquad \mathfrak{D}; \Gamma \vdash \sigma : \Delta \qquad \mathfrak{D}; \Gamma \vdash N : A[M/\Delta]}{\mathfrak{D}; \Gamma \vdash \ell\{\sigma\}\ N \triangleright M[\sigma, N/x] : B[\sigma, N/x]}$$

DCC judgements. Other judgements in DCC include well-formedness of contexts $\vdash \mathfrak{D}; \Gamma$, reductions $\mathfrak{D}; \Gamma \vdash M \triangleright N : A$, and the equivalence relation $\mathfrak{D}; \Gamma \vdash M \equiv N : A$.

DCC's dual context $\mathfrak{D}; \Gamma$ is well-formed when all types in Γ are well-typed (DCC-WF-TYPE), and every label is associated with well-typed data (DCC-WF-LABEL). In other words, if we have $\ell(\{\Delta\}, x{:}A \mapsto M : B)$, then M should have the type B judged under all previously defined labels $\mathfrak{D}$ (no self-reference) and the types $\Delta, x{:}A$ in its specification. Rule DCC-WF-TYPE allows the types in Γ to refer to the labels $\ell \in \mathfrak{D}$, but entries in $\mathfrak{D}$ cannot refer to variables $x \in \Gamma$ (DCC-WF-LABEL). The remaining rules are identical to their CC_ω counterparts apart from the dual context.

DCC

$$\textsc{dcc-wf-Empty}\ \frac{}{\vdash \cdot; \cdot} \qquad \textsc{dcc-wf-Label}\ \frac{\mathfrak{D}; \Delta, x{:}A \vdash M : B}{\vdash \mathfrak{D}, \ell(\{\Delta\}, x{:}A \mapsto N : B); \cdot} \qquad \textsc{dcc-wf-Type}\ \frac{\vdash \mathfrak{D}; \Gamma \qquad \mathfrak{D}; \Gamma \vdash A : U_i}{\vdash \mathfrak{D}; \Gamma, x{:}A}$$

Properties of DCC. Huang and Yallop [2023a] showed that the original DCC (with only Π-types and labels) is terminating and consistent. They defined a *backward transformation* from DCC to CC_ω by placing each label's definition back to the label's position as a lambda abstraction. Otherwise, the transformation simply maps Π-types to Π-types, applications to applications, etc. The backward transformation preserves reduction in lock-step and also preserves types (detailed in the extended version [Huang and Yallop 2023b]), so, DCC is terminating and consistent since CC_ω is.

The technical challenge in their proof lies in the treatment of functions/labels and applications. It is straightforward to extend the backward transformation to the extended DCC that maps Σ-types to Σ-types, unit to unit, etc. and show that the transformation remains type- and reduction-preserving, so that termination and consistency continue to hold for the extended version of DCC. The backward transformation also preserves equivalence, which implies non-confusion of type formers: for example, $\mathfrak{D}; \Gamma \vdash U_i \equiv \mathsf{Nat} : U$ implies that $U \equiv \mathit{Nat}$ in CC_ω, which leads to a contradiction.

However, Huang and Yallop did not establish the injectivity of type formers in their work, and it is not immediate that injectivity can be proved with DCC's known properties. Therefore, we will postulate injectivity since our focus is on deriving dependently typed abstract machines, not completing the meta-theory of DCC. We conjecture that injectivity can be proved by defining a logical relation for DCC judgements following [Abel et al. 2023, 2018; Danielsson et al. 2026].

Lemma 3.1 (Injectivity of type formers). *(Postulated) The type formers are injective in the following sense:*

- *If* $\mathfrak{D}; \Gamma \vdash \mathsf{U}_i \equiv \mathsf{U}_j : \mathsf{U}$, *then* $i = j$.
- *If* $\mathfrak{D}; \Gamma \vdash \Pi x{:}A.\, B \equiv \Pi x{:}A'.\, B' : \mathsf{U}$, *then* $\mathfrak{D}; \Gamma \vdash A \equiv A' : \mathsf{U}$ *and* $\mathfrak{D}; \Gamma, x{:}A \vdash B\ x \equiv B'\ x : \mathsf{U}$.
- *If* $\mathfrak{D}; \Gamma \vdash \Sigma x{:}A.\, B \equiv \Sigma x{:}A'.\, B' : \mathsf{U}$, *then* $\mathfrak{D}; \Gamma \vdash A \equiv A' : \mathsf{U}$ *and* $\mathfrak{D}; \Gamma, x{:}A \vdash B\ x \equiv B'\ x : \mathsf{U}$.

3.3 Dependent Abstract Machine

DAM

Instructions	I	::=	`var` x | `st` n | `ty` A | `app`
			| `clo` n ℓ | `pair` | `fst` | `snd`
			| `unit` | `lit` n | `suc` | `rec` P ℓ_z ℓ_s
Instruction sequences	ins	::=	`ret` | I; ins
Values	v	::=	n | $\langle\rangle$ | A | $\langle$v, v′$\rangle$ | ℓ{env}
Runtime environment/stack	env, st	::=	$\cdot$ | env :: v
Call frames	fr	::=	$\cdot$ | fr :: $\langle$ins, env, st$\rangle$
Code block	$\mathfrak{B}$	::=	$\cdot$ | $\mathfrak{B}; \ell(\{\Gamma\}, x{:}A \mapsto M : B) :$ ins
Machine configuration	C	::=	$\langle$ins, env, st, fr$\rangle_{\mathfrak{B}}$
Program			$\mathfrak{B}$; main$(\Gamma \vdash M : A) :$ ins

Fig. 7. Syntax of the dependent abstract machine (DAM)

Fig. 7 shows the syntax of the dependent abstract machine (DAM), a variant of the SECD machine. A DAM program consists of a list of code blocks $\mathfrak{B}$ and a main block. Each code block consists of an instruction sequence ins and a specification of its expected computation, which is written as a label context entry in DCC. A nice property of this defunctionalized setup is that the runtime values of DAM are exactly the closed values in DCC. Therefore, we can treat runtime environments and stacks as lists of DCC values and closing substitutions.

The instruction set of DAM is almost identical to that of the SECD machine, except that there is no `pushc` that could push an arbitrary piece of code during computation; all code is stored in $\mathfrak{B}$ and the only way to create closures is through instruction `clo` n ℓ that takes the top n stack items and forms a closure with label ℓ.

DAM's operational semantics has the form of $\langle \text{ins}, \text{env}, \text{st}, \text{fr}\rangle_{\mathfrak{B}} \longrightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}'\rangle_{\mathfrak{B}}$ where $\mathfrak{B}$ is a parameter that never changes (excerpt shown in Fig. 8). The operations of DAM have only a few differences from the SECD machine. `ty` A treats the current environment as a substitution and pushes the closed type A[env] on the stack. Application involves finding the code pointed to by the label on the list of code blocks $\mathfrak{B}$.

For the base case of recursors, when the top stack item is 0, the machine should load the instruction pointed to by label ℓ_z. However, since every label must point to a function, we encode the base case as a function with a dummy input of the unit type. We could also define a special class of labels whose pointed code can be loaded directly, but doing so complicates the syntax of DAM. At dam-red-RecZ, the machine loads the code pointed to by label ℓ_z on a new call frame, loads the environment with the current environment plus a dummy value $\langle\rangle$ and clears the stack. Other operations are identical to the rules for SECD machine we saw in §2.2.

DAM

Rule	Instruction	Env.	Stack	Frame	Condition
R	ins	env	st	fr	
	ins′	env′	st′	fr′	
	means that $\langle \text{ins}, \text{env}, \text{st}, \text{fr}\rangle_{\mathfrak{B}} \longrightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}'\rangle_{\mathfrak{B}}$ for rule DAM-RED-R.				
Ty	ty A ; ins	env	st	fr	
	ins	env	st :: A[env]	fr	
App	app ; ins	env	st :: ℓ{env′} :: v	fr	$\mathfrak{B}(\ell) = \text{ins}'$
	ins′	env′ :: v	·	fr :: ⟨ins, env, st⟩	
Clo	clo n ℓ	env	st :: st′	fr	$\lvert\text{st}'\rvert = n$
	ins′	env	st :: ℓ{st′}	fr	
RecZ	rec P ℓ_z ℓ_s ; ins	env	st :: 0	fr	$\mathfrak{B}(\ell_z) = \text{ins}'$
	ins′	env :: ⟨⟩	·	fr :: ⟨ins, env, st⟩	

Fig. 8. Operational semantics of DAM (excerpt)

3.4 Typing Judgements and Type Safety

In this section, we give typing judgements for well-typed instructions and well-formed configurations to show that DAM has type safety and partial correctness, using the same approach we saw from §2.3 to §2.5. Most of the judgements and the proofs for progress and preservation are similar to those for dependent SECD machine, and we only note the key differences here.

3.4.1 Instruction typing. The typing judgement for instruction (sequences) has the form $\mathfrak{D}; \Gamma \vdash \text{ins} : \sigma \rightarrow \sigma'$, where σ, σ' are abstract stacks in DCC. Again, we only consider well-typed abstract stacks in context $\mathfrak{D}; \Gamma$ for the typing judgement and we omit the requirement of σ, σ' being well-typed.

Similar to CC_ω, abstract stacks in DCC can be treated as substitutions, and we write $\mathfrak{D}; \Gamma \vdash \sigma : \Delta$ for env being viewed as a well-typed substitution from Δ to Γ and $\mathfrak{D}; \Gamma \vdash \sigma \equiv \sigma'$ for DCC's equivalence relation extended pointwise for abstract stacks.

Instruction `clo` n ℓ is well-typed if n matches the number of free variables required by ℓ and the top n items σ' match the types of free variables Δ in the specification of ℓ (DAM-TY-Clo). Application, along with most typing rules of DAM, remains almost identical to the corresponding rule for the dependent SECD machine, except for using DCC's dual context.

The recursor `rec` P ℓ_z ℓ_s takes three arguments: the return type P, a label ℓ_z pointing to the base case that produces M (with a dummy input of type Unit that M cannot access), and a label ℓ_s pointing to the inductive case that produces M′. This instruction consumes the natural number on top of the initial stack, leaving $\text{rec}_P(\text{M}, (\text{x}, \text{p}).\text{M}', \text{N})$ as the result (DAM-TY-Rec).

DAM

$$\textsc{dam-ty-Clo}\ \frac{|\Delta| = n \quad \mathfrak{D}; \Gamma \vdash \sigma' : \Delta \quad \ell(\{\Delta\}, x{:}A \mapsto M : B) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \mathsf{clo}\ n\ \ell : \sigma :: \sigma' \to \sigma :: \ell\{\sigma'\}}$$

$$\textsc{dam-ty-App}\ \frac{\mathfrak{D}; \Gamma \vdash M : \Pi x{:}A.\, B \quad \mathfrak{D}; \Gamma \vdash N : A}{\mathfrak{D}; \Gamma \vdash \mathsf{app} : \sigma :: M :: N \to \sigma :: M\ N}$$

$$\textsc{dam-ty-Rec}\ \frac{\begin{array}{c}\mathfrak{D}; \Gamma, x{:}\mathsf{Nat} \vdash P : U_i \quad \mathfrak{D}; \Gamma \vdash M : P[\mathsf{zero}/x] \quad \mathfrak{D}; \Gamma \vdash N : \mathsf{Nat} \\ \ell_z(\{\Gamma\}, _{:}\mathsf{Unit} \mapsto M : P[\mathsf{zero}/x]) \in \mathfrak{D} \\ \ell_s(\{\Gamma, x{:}\mathsf{Nat}, p{:}P \mapsto M' : P[\mathsf{suc}\ x/x]) \in \mathfrak{D}\end{array}}{\mathfrak{D}; \Gamma \vdash \mathsf{rec}\ P\ \ell_z\ \ell_s : \sigma :: N \to \sigma :: \mathsf{rec}_P(M, (x, p).M', N)}$$

3.4.2 Machine configuration and type safety. Given $\mathfrak{D}; \Gamma \vdash \mathsf{ins} : \sigma \to \sigma'$, the judgements for the configuration $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle_{\mathfrak{B}}$ to be well-formed are:

(1) $\boxed{\vdash \mathfrak{B} : \mathfrak{D}}$ The code blocks are well-typed with their type signatures $\mathfrak{D}$.
(2) $\boxed{\mathfrak{D} \vdash \mathsf{env} \vDash \Gamma}$ The runtime environment has types Γ.
(3) $\boxed{\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{st} \vDash \sigma}$ The runtime stack implements σ, given some env that implements Γ.
(4) $\boxed{\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{fr} : [M : A]}$ The stack of call frames is well-formed, and the instruction of the top frame expects its callee to return an abstract term M.

Since DAM's runtime values are exactly closed values in DCC, we can use DCC's typing judgement as the judgement for runtime values.

Code blocks. As the syntax suggests, the type signature of each code block is exactly a label entry in DCC, so, $\mathfrak{B}$ is specified with a label context $\mathfrak{D}$. The judgement $\vdash \mathfrak{B} : \mathfrak{D}$ means that the instruction in each code block is well-typed with respect to its signature. An instruction sequence implements a defunctionalized label if the instructions eventually produce the specified term on top of the stack, starting with an empty stack.

DAM

$$\textsc{dam-blocks-Nil}\ \frac{}{\vdash \cdot : \cdot} \qquad \textsc{dam-blocks-Cons}\ \frac{\vdash \mathfrak{B} : \mathfrak{D} \quad \mathfrak{D}; \Delta, x{:}A \vdash M : B \quad \mathfrak{D}; \Delta, x{:}A \vdash \mathsf{ins} : \cdot \to \sigma :: M}{\vdash \mathfrak{B}; \ell(\{\Delta\}, x{:}A \mapsto M : B) : \mathsf{ins} : \mathfrak{D}, \ell(\{\Delta\}, x{:}A \mapsto M : B)}$$

Environment, stack, and frames. The rest of the judgements are again similar to those in dependent SECD. The judgement $\mathfrak{D} \vdash \mathsf{env} \vDash \Gamma$ means that env is a concrete implementation of context Γ, and we can treat env as a closing substitution for Γ. The empty case of this judgement is trivial, and $\mathfrak{D} \vdash \mathsf{env} :: v \vDash \Gamma, x{:}A$ if $\mathfrak{D} \vdash \mathsf{env} \vDash \Gamma$ and v has type $A[\mathsf{env}]$.

The judgement for runtime stacks $\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{st} \vDash \sigma$ means that st implements the abstract stack σ under runtime environment $\mathsf{env} : \Gamma$. The empty case requires $\mathfrak{D} \vdash \mathsf{env} \vDash \Gamma$, and $\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{st} :: v \vDash \sigma :: M$ if $\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{st} \vDash \sigma$ and $\mathsf{val} \equiv M[\mathsf{env}]$, i.e. the value implements the term M closed by env. The rules for well-formed call frames are the same ones in dependent SECD with all judgements replaced with their counterparts in DCC/DAM.

DAM

$$\textsc{dam-env-Cons}\ \frac{\mathfrak{D} \vdash \mathsf{env} \vDash \Gamma \quad \mathfrak{D} \vdash v : A[\mathsf{env}]}{\mathfrak{D} \vdash \mathsf{env}, v \vDash \Gamma, x{:}A} \qquad \textsc{dam-st-Cons}\ \frac{\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{st} \vDash \sigma \quad \mathfrak{D} \vdash v : A[\mathsf{env}] \quad \mathfrak{D}; \Gamma \vdash M : A \quad \mathfrak{D} \vdash M[\mathsf{env}] \equiv v : A[\mathsf{env}]}{\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{st}, v \vDash \sigma :: M}$$

A configuration is well-formed if its code blocks are well-typed according to their specified signatures, the instruction sequence is well-typed, and the runtime environment, stack, and call frames are compatible with the instructions and with each other. As discussed in §2.4, conditions (6) and (7) make sure that the return value of the current instruction sequence is compatible with the expected return value on the top of the call frame. We also require that labels for base cases of recursors does not access the dummy variable.

We can also show that the parameters in DAM's typing judgements are well-formed by induction and closed values are in canonical forms by non-confusion of DCC. We then prove progress and preservation by induction on the instruction sequence; details of the proofs are similar to the proofs for dependent SECD machine. From type safety we conclude that DAM has partial correctness.

Definition 3.2 (Configuration well-formedness). *A configuration* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle_{\mathfrak{B}}$ *is well-formed, denoted as* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle_{\mathfrak{B}}$*, if the following conditions hold:*

(1) $\vdash \mathfrak{B} : \mathfrak{D}$, *where* $\mathfrak{D}$ *is the label context of* $\mathfrak{B}$.
(2) $\mathfrak{D}; \Gamma \vdash \mathsf{ins} : \sigma \longrightarrow \sigma' :: \mathsf{M}$, *for some* Γ, σ, σ', *and* M, *with* $\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A}$ *for some* A.
(3) $\mathfrak{D} \vdash \mathsf{env} \vDash \Gamma$.
(4) $\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{st} \vDash \sigma$ *for some* Δ.
(5) $\mathfrak{D}; (\mathsf{env}' : \Gamma') \vdash \mathsf{fr} : [\mathsf{M}' : \mathsf{A}']$, *for some* env', Γ', M', *and* A'.
(6) $\mathfrak{D} \vdash \mathsf{A}[\mathsf{env}] \equiv \mathsf{A}'[\mathsf{env}'] : \mathsf{U}$.
(7) $\mathfrak{D} \vdash \mathsf{M}[\mathsf{env}] \equiv \mathsf{M}'[\mathsf{env}'] : \mathsf{A}[\mathsf{env}]$.
(8) *For all labels* ℓ *used in base case of recursors,* $\mathfrak{B}(\ell)$ *do not use the* Unit*-typed dummy variable.*

Lemma 3.3 (Canonicity of values). *If* $\mathfrak{D}; \cdot \vdash \mathsf{v} : \mathsf{A}$, *then*

- *If* $\mathfrak{D} \vdash \mathsf{A} \equiv \mathsf{U}_i : \mathsf{U}_{i+1}$, *then* $\mathsf{v} = \mathsf{A}'$ *for some type* A'.
- *If* $\mathfrak{D} \vdash \mathsf{A} \equiv \mathit{Nat} : \mathsf{U}$, *then* $\mathsf{v} = n$ *for some natural number* n.
- *If* $\mathfrak{D} \vdash \mathsf{A} \equiv \mathit{Unit} : \mathsf{U}$, *then* $\mathsf{v} = \langle\rangle$.
- *If* $\mathfrak{D} \vdash \mathsf{A} \equiv \Pi \mathsf{x}{:}\mathsf{A}'.\,\mathsf{B}' : \mathsf{U}$, *then* $\mathsf{v} = \ell\{\mathsf{env}\}$ *for some* env, ℓ.
- *If* $\mathfrak{D} \vdash \mathsf{A} \equiv \Sigma \mathsf{x}{:}\mathsf{A}'.\,\mathsf{B}' : \mathsf{U}$, *then* $\mathsf{v} = \langle \mathsf{v}, \mathsf{v}'\rangle$ *for some* v, v'.

Theorem 3.4 (Progress). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle_{\mathfrak{B}}$, *then either* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle_{\mathfrak{B}} \longrightarrow \langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}'\rangle_{\mathfrak{B}}$ *or* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle_{\mathfrak{B}}$ *halts.*

Theorem 3.5 (Preservation). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle_{\mathfrak{B}}$ *and* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle_{\mathfrak{B}} \longrightarrow \langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}'\rangle_{\mathfrak{B}}$, *then* $\vdash \langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}'\rangle_{\mathfrak{B}}$.

Corollary 3.6 (Type safety). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle_{\mathfrak{B}}$, *then it never reduces to a configuration that is irreducible and not at halt.*

Corollary 3.7 (Partial correctness). *If* $\mathfrak{D}; \Gamma \vdash \mathsf{ins} : \sigma \longrightarrow \sigma' :: \mathsf{M}$ *with* $\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A}$, *then for all* $(\mathfrak{B}, \mathsf{env}, \mathsf{st})$ *such that* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \cdot\rangle_{\mathfrak{B}}$ *and* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \cdot\rangle_{\mathfrak{B}} \Downarrow \mathsf{v}$, *we have* $\mathfrak{D} \vdash \mathsf{v} \equiv \mathsf{M}[\mathsf{env}] : \mathsf{A}[\mathsf{env}]$.

3.5 Termination and Total Correctness

We prove termination for DAM by defining a backward transformation to the dependent SECD machine that preserves types and reductions. The transformation for the specifications is based on the backward transformation $[\![-]\!]_b$ from well-typed DCC terms to CC_ω by Huang and Yallop. $[\![\mathsf{M}]\!]_b$ implicitly takes the well-typed derivation of M an argument. It maps $\ell\{\sigma\}$ to $\lambda x{:}([\![\mathsf{A}]\!]_b[[\![\sigma]\!]_b]).[\![\mathsf{N}]\!]_b[[\![\sigma]\!]_b]$ given $\ell(\{\Delta\}, \mathsf{x}{:}\mathsf{A} \mapsto \mathsf{N} : \mathsf{B})$ in the label context. Otherwise, the transformation proceeds recursively into subterms, maps DCC applications to CC_ω applications, and so on. The transformation extends to contexts pointwise, and it is type-preserving in the sense of Lemma 3.8.

Lemma 3.8 ($[\![-]\!]_b$ preserves types). *If* $\mathfrak{D}; \Gamma \vdash M : A$, *then* $[\![\Gamma]\!]_b \vdash [\![M]\!]_b : [\![A]\!]_b$.

$[\![\mathrm{DAM}]\!]_{cf}$

$$[\![\mathtt{clo}\ n\ \ell]\!]_i = \mathtt{clo}\ n\ [\![\mathfrak{B}(\ell)]\!]_i \quad [\![\mathtt{rec}\ \mathsf{P}\ \ell_z\ \ell_s]\!]_i = \mathtt{rec}\ [\![\mathsf{P}]\!]_b\ [\![\mathfrak{B}(\ell_z)]\!]_i\ [\![\mathfrak{B}(\ell_s)]\!]_i$$
$$[\![\ell\{\overline{M}\}]\!]_v = \langle [\![\overline{M}]\!]_v, [\![\mathfrak{B}(\ell)]\!]_i \rangle \quad [\![A]\!]_v = [\![A]\!]_b \quad [\![\langle v, v' \rangle]\!]_v = \langle [\![v]\!]_v, [\![v']\!]_v \rangle$$
$$[\![\mathsf{fr} :: \langle \mathsf{ins}, \mathsf{env}, \mathsf{st} \rangle]\!]_{fr} = [\![\mathsf{fr}]\!]_{fr} :: \langle [\![\mathsf{ins}]\!]_i, [\![\mathsf{env}]\!]_v, [\![\mathsf{st}]\!]_v \rangle$$
$$[\![\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle_{\mathfrak{B}}]\!]_{cf} = \langle [\![\mathsf{ins}]\!]_i, [\![\mathsf{env}]\!]_v, [\![\mathsf{st}]\!]_v, [\![\mathsf{fr}]\!]_{fr} \rangle$$

Fig. 9. Backward transformation from DAM to dependent SECD (excerpt)

Fig. 9 shows the backward transformation $[\![-]\!]_i$ on instructions that stitches each label's associated code back to the label's location, and other instructions remain the same. We also define a function $[\![-]\!]_v$ that transforms DAM values to SECD-machine values. Natural numbers and the unit value remain the same. $[\![-]\!]_v$ places back the code pointed by the label in closures, returns a closed CC_ω type with an empty environment for types, and maps recursively for pairs. We can transform DAM environments and stacks by applying $[\![-]\!]_v$ pointwise and transform call frames with a combination of $[\![-]\!]_i$ and $[\![-]\!]_v$. Finally, we can transform a DAM configuration by applying the respective backward transformations for the four components of the machine. Transformations of the machine $[\![-]\!]_{cf}$ and its components $[\![-]\!]_i$, $[\![-]\!]_v$, and $[\![-]\!]_{fr}$ take the well-formedness proof of the machine as an implicit argument.

The backward transformation from DAM to dependent SECD is type-preserving: a well-formed DAM configuration transforms to a well-formed SECD machine configuration (Lemma 3.9). We prove type preservation by proving the following sub-goals by induction:

- If $\mathfrak{D}; \Gamma \vdash \mathsf{ins} : \sigma \longrightarrow \sigma'$, then $[\![\Gamma]\!]_b \vdash [\![\mathsf{ins}]\!]_i : [\![\sigma]\!]_b \longrightarrow [\![\sigma']\!]_b$.
- If $\mathfrak{D} \vdash v : A$, then $[\![v]\!]_v \vDash [\![v]\!]_b : [\![A]\!]_b$.
- If $\mathfrak{D} \vdash \mathsf{env} \vDash \Gamma$, then $[\![\mathsf{env}]\!]_v \vDash [\![\mathsf{env}]\!]_b : [\![\Gamma]\!]_b$.
- If $\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{st} \vDash \sigma$, then $[\![\mathsf{env}]\!]_v :_{[\![\mathsf{env}]\!]_b} [\![\Gamma]\!]_b \vdash [\![\mathsf{st}]\!]_v \vDash [\![\sigma]\!]_b$.
- If $\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{fr} : [M : A]$, then $[\![\mathsf{env}]\!]_v :_{[\![\mathsf{env}]\!]_b} [\![\Gamma]\!]_b \vdash [\![\mathsf{fr}]\!]_{fr} : [[\![M]\!]_b : [\![A]\!]_b]$.

Note that there are two ways to transform a DAM value: as a CC_ω term or as a dependent SECD machine value. Type preservation requires that the two transformations are coherent, i.e. $[\![v]\!]_v$ implements $[\![v]\!]_b$.

We also prove that the backward transformation preserves reductions: if a DAM configuration steps to some next state $\langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}' \rangle_{\mathfrak{B}}$, then the transformed SECD machine configuration steps the transformation of $\langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}' \rangle_{\mathfrak{B}}$ (Lemma 3.10). Combined with type-preservation, we can prove termination for all well-formed DAM programs by contradiction (Theorem 3.11), and finally conclude that DAM has total correctness (Corollary 3.12) from partial correctness and termination.

Lemma 3.9 ($[\![-]\!]_{cf}$ preserves well-formedness). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle_{\mathfrak{B}}$, *then* $\vdash [\![\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle_{\mathfrak{B}}]\!]_{cf}$.

Lemma 3.10 ($[\![-]\!]_{cf}$ preserves reductions). *If* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle_{\mathfrak{B}} \longrightarrow \langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}' \rangle_{\mathfrak{B}}$, *then* $[\![\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle_{\mathfrak{B}}]\!]_{cf} \longrightarrow [\![\langle \mathsf{ins}', \mathsf{env}', \mathsf{st}', \mathsf{fr}' \rangle_{\mathfrak{B}}]\!]_{cf}$.

Theorem 3.11 (Termination). *If* $\vdash \langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle_{\mathfrak{B}}$, *then* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle_{\mathfrak{B}} \Downarrow v$.

Corollary 3.12 (Total correctness). *If* $\mathfrak{D}; \Gamma \vdash \mathsf{ins} : \sigma \longrightarrow \sigma' :: M$ *with* $\mathfrak{D}; \Gamma \vdash M : A, \vdash \mathfrak{B} : \mathfrak{D}$, $\mathfrak{D} \vdash \mathsf{env} \vDash \Gamma$, *and* $\mathfrak{D}; (\mathsf{env} : \Gamma) \vdash \mathsf{st} \vDash \sigma$, *then* $\langle \mathsf{ins}, \mathsf{env}, \mathsf{st}, \cdot \rangle_{\mathfrak{B}} \Downarrow v$ *with* $\mathfrak{D} \vdash v \equiv M[\mathsf{env}] : A[\mathsf{env}]$.

Corollary 3.13 (Total correctness for programs). *If* $\mathfrak{D}; \cdot \vdash \mathsf{ins} : \cdot \longrightarrow \sigma :: M$ *with* $\mathfrak{D} \vdash M : A$, *then* $\langle \mathsf{ins}, \cdot, \cdot, \cdot \rangle_{\mathfrak{B}} \Downarrow v$ *with* $\mathfrak{D} \vdash v \equiv M : A$.

3.6 Compilation

Finally, we define a type-preserving code generation scheme from DCC to DAM (Fig. 10). Given a DCC derivation, $\mathsf{gen}(\mathfrak{D}; \Gamma \vdash M : A)$ produces an instruction sequence for M and code blocks $\mathfrak{B}$ for $\mathfrak{D}$. Code generation is syntactic except for recursors: it maps variables to `var`, applications to `app`, type constants to `ty`, etc., and produces $\mathfrak{B}$ by generating code for each label in $\mathfrak{D}$.

gen(DCC)

$$
\begin{aligned}
[\![\ell\{M_1, \cdots, M_n\}]\!] &= [\![M_1]\!]; \cdots; [\![M_n]\!]; \mathtt{clo}\ n\ \ell \\
[\![\mathsf{rec}_P(M, (x, p).M', N)]\!] &= \mathtt{rec}\ P\ \ell_z\ \ell_s \\
[\![\mathfrak{D}, \ell(\{\Delta\}, x{:}A \mapsto M : B)]\!] &= [\![\mathfrak{D}]\!]; \ell(\{\Delta\}, x{:}A \mapsto M : B) : [\![M]\!] \\
\mathsf{gen}(\mathfrak{D}; \Gamma \vdash M : A) &= ([\![M]\!], [\![h_D(\mathfrak{D}); h_{exp}(M)]\!])
\end{aligned}
$$

Fig. 10. Code generation for DCC (excerpt)

Generating label for recursors. We assume that the input DCC programs are obtained by defunctionalizing source programs in CC_ω, using an extended version of Huang and Yallop [2023b]'s defunctionalization which does not generate labels for recursor cases. We therefore define functions h_D and h_{exp} to hoist the code for all recursor sub-terms in the body of labels in $\mathfrak{D}$ and in an expression M, generating an expanded label context $\mathfrak{D}'$.

Hoisting is simple: for expressions, h_{exp} traverses the syntax, hoisting two labels ℓ_z, ℓ_s when it reaches a recursor; for the label context, h_D inserts the hoisted recursor cases from the body of each label before that label. We only hoist recursor cases found in the bodies of labels because only those cases are involved in execution; the role of recursors in the type context Γ, for example, vectors encoded as iterated pairs, is limited to specification. For the same reason, we extend DCC with recursors taking expressions instead of labels, so that we can avoid generating computationally irrelevant code for recursors that exist only for specification purposes.

gen(DCC)

$$
\begin{aligned}
h_{exp}(\mathsf{rec}_P(M, (x, p).M', N)) = h_{exp}(M);\ & h_{exp}(M');\ h_{exp}(N); \\
& \ell_z(\{\Gamma\}, _{:}\mathsf{Unit} \mapsto M : P[\mathsf{zero}/x]); \\
& \ell_s(\{\Gamma, x{:}\mathsf{Nat}\}, p{:}P\ x \mapsto M' : P[\mathsf{suc}\ x/x]) \\
h_D(\mathfrak{D}, \ell(\{\Delta\}, x{:}A \mapsto M : B)) = h_D(\mathfrak{D});\ & h_{exp}(M); \ell(\{\Delta\}, x{:}A \mapsto M : B)
\end{aligned}
$$

Fig. 11. Recursor hoisting (excerpt)

Type preservation. The code generation is type-preserving: $\mathsf{gen}(\mathfrak{D}; \Gamma \vdash M : A)$ produces an instruction sequence that computes M and a well-formed code block $\mathfrak{B}$ that implements $\mathfrak{D}$ (Theorem 3.17). It is also correct for whole-program and separate compilation as a corollary of total correctness.

We first define the notion of having recursor labels for a recursor term (Definition 3.14). We then prove two lemmas stating that hoisting produces a well-formed label context that extends the original context and has recursor labels for all recursor sub-terms (Lemma 3.15), and that code generation produces well-typed results if the label context that has labels for all recursors (Lemma 3.16). As a corollary, code generation is type-preserving in the sense of Theorem 3.17, and it is correct for whole-program and separate compilations since DAM has total correctness.

Definition 3.14. *$\mathfrak{D}$ has recursor labels for a recursor $\mathfrak{D}; \Delta \vdash rec_P(M, (x, p).M', N) : P[N/x]$ if there exist $\ell_z(\{\Delta\}, x{:}Unit \mapsto M : P[zero/x]) \in \mathfrak{D}$ and $\ell_s(\{\Delta, x{:}Nat\}, p{:}P \mapsto M' : P[suc\ x/x]) \in \mathfrak{D}$.*

Lemma 3.15. *Suppose $\mathfrak{D}; \Gamma \vdash M : A$, and let $\mathfrak{D}' = h_D(\mathfrak{D})$; $h_{exp}(M)$. Then $\vdash \mathfrak{D}'$, $\mathfrak{D} \subseteq \mathfrak{D}'$, and $\mathfrak{D}'$ has recursor labels for all recursor sub-terms in $\mathfrak{D}$ and M.*

Lemma 3.16. *If $\mathfrak{D}; \Gamma \vdash M : A$ and $\mathfrak{D}$ has recursor labels for all recursor sub-terms in $\mathfrak{D}$ and M, then $\mathfrak{D}; \Gamma \vdash [\![M]\!] : \cdot \longrightarrow \sigma :: M$ and $\vdash [\![\mathfrak{D}]\!] : \mathfrak{D}$.*

Proof. By induction over the typing derivation of DCC. □

Theorem 3.17 (Type preservation). *Suppose $\mathfrak{D}; \Gamma \vdash M : A$, and let $(ins, \mathfrak{B}) = gen(\mathfrak{D}; \Gamma \vdash M : A)$. Then $\vdash \mathfrak{B} : \mathfrak{D}'$ and $\mathfrak{D}'; \Gamma \vdash ins : \cdot \longrightarrow \sigma :: M$.*

Corollary 3.18 (Correctness of whole-program compilation). *Suppose $\mathfrak{D}; \cdot \vdash M : A$, and let $(ins, \mathfrak{B}) = gen(\mathfrak{D}; \cdot \vdash M : A)$. Then, $\langle ins, \cdot, \cdot, \cdot \rangle_{\mathfrak{B}} \Downarrow v$ with $\mathfrak{D} \vdash v \equiv M : A$.*

Corollary 3.19 (Correctness of separate compilation). *Suppose $\mathfrak{D}; \Gamma \vdash M : A$, and let env be a list of closed DCC values such that $\mathfrak{D} \vdash env : \Gamma$ and $(ins, \mathfrak{B}) = gen(\mathfrak{D}; \Gamma \vdash M : A)$. Then, $\langle ins, env, \cdot, \cdot \rangle_{\mathfrak{B}} \Downarrow v$ with $\mathfrak{D} \vdash v \equiv M[env] : A[env]$.*

4 Conclusion and Discussion

We have introduced a principle for developing dependently typed abstract machines. We apply the principle to develop two machines — the dependent SECD machine and the TAL-style dependent abstract machine (DAM), both enjoy *total correctness*: every well-typed machine halts with exactly the value specified by its type. We give a type-preserving translation from DCC to DAM which, combined with existing work on defunctionalization from CC_ω, yields the first type-preserving compiler for a fully dependently typed language, with whole-program and separate compilation correctness checkable by type checking the compiled instructions, independently of the compiler or source code that produced them. We have implemented both machines in Agda and formalized total correctness for the dependent SECD machine and type safety for DAM.

4.1 Related work

Typed assembly languages. The most well-known and influential work on typed assembly language is TAL [Morrisett et al. 1998], which serves as a compilation target for System F. The authors subsequently developed several variants of TAL, including a variant based on a stack machine [Morrisett et al. 2002], a variant for x86 [Crary et al. 1999] that served as a compilation target for a C-like language, and a foundational language TALT that supported more sophisticated features such as heterogeneous tuples and disjoint sums [Crary 2003]. Chen et al. [2003] extend the idea of TAL to a language LTAL (low-level typed assembly language) with provable soundness, i.e. with the property that if an LTAL program type-checks it is safe to execute.

§3.1 touched on the dependently-typed assembly languages Singleton [Winwood and Chakravarty 2011] and DTAL [Xi and Harper 2001], both of which relate runtime values and compile-time types through singleton types. Our abstract stacks could be considered a form of singleton that relates runtime and compile-time objects in a clean and principled way with the specification language, whereas DTAL relies on a constraint solver and Singleton uses a mixture of low-level register types and dependently typed logical types.

Shao et al. [2002] used a similar two-level type system, where the specification language is the Calculus of Inductive Constructions and the target language is System F, to certify CPS transformation, closure conversion, and optimizations such as bound check elimination in the target. In a

different vein, earlier work by Ritter [1994] studies the assignment of fully dependent types to the combinators of a Categorical Abstract Machine [Cousineau et al. 1987].

The TAL family enjoys type safety, i.e. the promise that well-typed programs will not get stuck. Wright and Felleisen [1994] refer to the property as *weak soundness*, defining *strong soundness* as type safety with termination, so that well-typed programs always halt with a value of the desired type. Our total correctness property provides the stronger guarantee that the output value is exactly the one specified by its singleton type.

Type preserving compilation. Preserving types partway through compilation is common in the compilation of functional programming languages: for example, the Glasgow Haskell Compiler (GHC) uses a typed intermediate language (IL) that extends System F [Sulzmann et al. 2007], and the whole-program Standard ML compiler MLton makes effective use of typed ILs to perform unboxing, data flattening and other optimizations [Weeks 2006]. Compilers that preserve types all (or almost all) the way are rarer. The Standard ML compiler TIL [Tarditi et al. 1996] preserves types through several ILs most of the way down to machine code; TAL, the typed assembly language presented in §3.1, subsequently pushed the approach even further. Drawing inspiration from TAL, Chen et al. [2008] present Bartok, a type-preserving compiler for CIL, an IR for C# code. Guillemette and Monnier [2006, 2008] take a different approach in a type-preserving compiler for System F, representing programs using Haskell's GADTs and type families, then using GHC's type checker to check the correctness of intermediate results.

The extension to dependent types of type-preserving transformations that are commonly used in the compilation of functional languages introduces various challenges. Bowman and Ahmed [2018] study closure conversion for CC, using an abstract definition of closures that draws inspiration from Minamide et al. [1996]. The abstract definition avoids problems with parametricity, impredicativity and erasure that arise when naively extending the usual approach of using existential types to the dependent setting. Similarly, Huang and Yallop [2023a] use a variant of defunctionalization with abstract labels (presented in §3.2 of the present work) that avoids the problems with universe checking, positivity checking and termination that arise in attempting to transfer the work of Pottier and Gauthier [2004] on polymorphic defunctionalization to a dependently typed setting.

In a study of the CPS conversion for CC, Bowman et al. [2018] show that, despite previous results by Barthe and Uustalu [2002], type-preserving CPS for CC is achievable with an additional typing rule in an extensional context. Koronkevich et al. [2022] similarly rely on extensionality to reflect propositional equalities that record the definitions of let-bound variables in a type-preserving translation of Extended CC into ANF. Cong et al. [2024] revisit Bowman et al.'s work on CPS conversion, showing that let-binding continuations avoids the need for a non-standard typing rule. We believe that our development of DAM from DCC could be applied to the ANF transformation, using the two-level typing to assign singleton types to the let-bound variables and subsequently to derive a corresponding abstract machine.

Verified compilation. There are other approaches besides type preservation to improving confidence in the correctness of compilation. In particular, verifying the compiler itself in a proof assistant can formally establish that each compilation phase preserves the semantics (but not necessarily the types) of programs. Notable examples of this approach include CompCert, a C compiler written in Rocq [Leroy 2009], CertiRocq [Paraskevopoulou et al. 2021], a compiler for Rocq's specification language Gallina, which erases types and proofs before compiling through untyped intermediate languages to C, and CakeML, a Standard ML compiler written in HOL4 [Tan et al. 2019], which has recently been extended to support verified compilation of a Haskell-like language [Kanabar et al. 2023] and of a subset of Scheme [Lasnier et al. 2026]. Although verified compilation and type-preserving compilation look quite different for the simply-typed (or polymorphic) source

languages, they effectively converge for dependently typed languages, where showing that types are preserved necessitates also showing that reduction is preserved, as Bowman et al. [2018] first observed. Our type-preserving compilation from DCC to DAM is thus an instance of so-called *translation validation* [Pnueli et al. 1998], in which the correspondence of the semantics of the source and target programs can be straightforwardly mechanically checked after compilation. Verified compilation and translation validation offer similar guarantees — every compiled (respectively, validated) program is correct, given trust in the proof assistant or in the validator — but differ in practice. Our form of translation validation requires a type system for the target language rather than only a semantics, and accepts only outputs definitionally equal to their specification (§4.2); in return, the compiler itself need not be verified, since type checking its output suffices.

4.2 Limitations and Future Work

Registers, effects, and multi-pass compilers. Our framework applies to pure, terminating source languages and abstract machines. Generalization to effectful systems and register machines poses new challenges, such as the need to design typing rules for effectful instructions that modify register contents and to correspondingly extend the termination proof to show consistency of the dependent type theory.

In the other direction, the work here might be connected with previous work on type-preserving compilation (e.g. Bowman et al. [2018]; Cong et al. [2024]; Koronkevich et al. [2022]) to build realistic compiler pipelines, extending our two-phase pipeline that composes defunctionalization and translation to DAM. Our approach imposes several requirements: the language from which the final machine is derived serves as its specification language, each pass must be type-preserving and correct, and the specification language must be normalising, consistent, and have injective and non-confusable type formers (§§2.1 and 2.5).

Remaining inefficiencies. The DAM eliminates the inefficiency in the dependent SECD machine that arises from retaining well-formedness proofs of runtime environments for executing the $\mathtt{ty}\ A$ instruction. However, it does not eliminate the similar inefficiency that arises from retaining computationally irrelevant types until run time.

We conjecture that defunctionalization also works with Quantitative Type Theory (QTT) [Atkey 2018], and that our whole compilation scheme can be similarly extended with quantitative types. QTT offers precise annotations that support type erasure and linearity, and we hope to use it to develop safer and more efficient target code in the future.

Linking and optimization. In principle, dependently typed assembly should support safe linking and type-preserving optimizations, but we have not yet investigated either. With the current design, source code changes might cause linking to fail, and the optimizations are required to respect definitional equalities. For example, our current design cannot verify the optimization of $(x + 2) + 3$ to $x + 5$ if addition is defined by induction on the left argument, and re-defining addition to match on the right argument could break linking with previously compiled components that require the former definition of addition.

We plan to extend the framework to support a range of data types beyond the unit type and natural numbers, including identity types and inductive families [Dybjer 1994] more generally. The framework is designed to allow incorporating inductive families with syntax, typing rules and operational semantics similar to our existing rules for natural numbers.

The addition of the identity type is a promising route to extending support for safe linking and type-preserving optimizations, because it allows expressing proofs of program equivalence that go beyond purely definitional equality.

Disclosure of Use of AI in Research

The research and artifact development was carried out without the use of AI, with one exception: we used Claude Fable 5.1 to produce an initial Agda implementation of the termination proof based on our Agda development and the sketch in the extended paper.

Data Availability Statement

We have mechanized the proofs for type safety and termination of the dependent SECD machine and type safety for DAM in Agda, and the mechanized proofs are available as an artifact [Huang and Yallop 2026].

Acknowledgments

We thank Marcelo Fiore, Meven Lennon-Bertrand, András Kovács, and Amal Ahmed for insightful discussions and the anonymous reviewers for helpful suggestions and feedback.

References

Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP (2023), 920–954. https://doi.org/10.1145/3607862

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.* 2, POPL (2018), 23:1–23:29. https://doi.org/10.1145/3158111

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. https://doi.org/10.1145/3209108.3209189

Gilles Barthe and Tarmo Uustalu. 2002. CPS translating inductive and coinductive types. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02), Portland, Oregon, USA, January 14-15, 2002*, Peter Thiemann (Ed.). ACM, 131–142. https://doi.org/10.1145/503032.503043

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 73–78. https://doi.org/10.1007/978-3-642-03359-9_6

William J Bowman. 2019. *Compiling with dependent types.* Ph.D. Dissertation. Northeastern University.

William J. Bowman and Amal Ahmed. 2018. Typed closure conversion for the calculus of constructions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 797–811. https://doi.org/10.1145/3192366.3192372

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2018. Type-preserving CPS translation of Σ and Π types is not not possible. *Proc. ACM Program. Lang.* 2, POPL (2018), 22:1–22:33. https://doi.org/10.1145/3158110

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPICS.ECOOP.2021.9

Juan Chen, Chris Hawblitzel, Frances Perry, Michael Emmi, Jeremy Condit, Derrick Coetzee, and Polyvios Pratikakis. 2008. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 183–192. https://doi.org/10.1145/1375581.1375604

Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. 2003. A provably sound TAL for back-end optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 208–219. https://doi.org/10.1145/781131.781155

Youyou Cong, Hironori Kawazoe, and Hidehiko Masuhara. 2024. A Mostly CPS, Partly ANF Translation of Dependent Types. In *Proceedings of the 36th Symposium on Implementation and Application of Functional Languages, IFL 2024, Nijmegen, The Netherlands, August 26-28, 2024.* ACM, 65–74. https://doi.org/10.1145/3723325.3723341

Thierry Coquand. 1986. An Analysis of Girard's Paradox. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986.* IEEE Computer Society, 227–236.

Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. 1987. The Categorical Abstract Machine. *Sci. Comput. Program.* 8, 2 (1987), 173–202. https://doi.org/10.1016/0167-6423(87)90020-7

Karl Crary. 2003. Toward a foundational typed assembly language. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 198–212. https://doi.org/10.1145/604131.604149

K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. 1999. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*. 25–35.

Nils Anders Danielsson, Naïm Camille Favier, and Ondrej Kubánek. 2026. Normalisation for First-Class Universe Levels. *Proc. ACM Program. Lang.* 10, POPL (2026), 64–88. https://doi.org/10.1145/3776645

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

Peter Dybjer. 1994. Inductive Families. *Formal Aspects Comput.* 6, 4 (1994), 440–465. https://doi.org/10.1007/BF01211308

Herman Geuvers. 1994. A short and flexible proof of Strong Normalization for the Calculus of Constructions. In *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers (Lecture Notes in Computer Science, Vol. 996)*, Peter Dybjer, Bengt Nordström, and Jan M. Smith (Eds.). Springer, 14–38. https://doi.org/10.1007/3-540-60579-7_2

Neal Glew and J. Gregory Morrisett. 1999. Type-Safe Linking and Modular Assembly Language. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 250–261. https://doi.org/10.1145/292540.292563

Healfdene Goguen. 1994. *A typed operational semantics for type theory.* Ph.D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/405

Louis-Julien Guillemette and Stefan Monnier. 2006. Type-Safe Code Transformations in Haskell. In *Proceedings of the Programming Languages meets Program Verification, PLPV@IJCAR 2006, Part of FLoC 2006, Seattle, WA, USA, August 21, 2006 (Electronic Notes in Theoretical Computer Science, Vol. 174)*, Aaron Stump and Hongwei Xi (Eds.). Elsevier, 23–39. https://doi.org/10.1016/J.ENTCS.2006.10.036

Louis-Julien Guillemette and Stefan Monnier. 2008. A type-preserving compiler in Haskell. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 75–86. https://doi.org/10.1145/1411204.1411218

Robert Harper and Mark Lillibridge. 1993. Explicit Polymorphism and CPS Conversion. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 206–219. https://doi.org/10.1145/158511.158630

Yulong Huang and Jeremy Yallop. 2023a. Defunctionalization with Dependent Types. *Proc. ACM Program. Lang.* 7, PLDI (2023), 516–538. https://doi.org/10.1145/3591241

Yulong Huang and Jeremy Yallop. 2023b. Defunctionalization with Dependent Types: Extended version. arXiv:2304.04574 [cs.PL] https://arxiv.org/abs/2304.04574

Yulong Huang and Jeremy Yallop. 2026. *Fully Dependent Abstract Machines: Agda Implementation.* https://doi.org/10.5281/zenodo.23314099

Suresh Jagannathan, Stephen Weeks, and Andrew K. Wright. 1997. Type-Directed Flow Analysis for Typed Intermediate Languages. In *Static Analysis, 4th International Symposium, SAS '97, Paris, France, September 8-10, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1302)*, Pascal Van Hentenryck (Ed.). Springer, 232–249. https://doi.org/10.1007/BFB0032745

Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola, and Riccardo Zanetti. 2023. PureCake: A Verified Compiler for a Lazy Functional Language. *Proc. ACM Program. Lang.* 7, PLDI (2023), 952–976. https://doi.org/10.1145/3591259

Paulette Koronkevich, Ramon Rakow, Amal Ahmed, and William J. Bowman. 2022. ANF preserves dependent types up to extensional equality. *J. Funct. Program.* 32 (2022), e12. https://doi.org/10.1017/S0956796822000090

András Kovács. 2020. Elaboration with first-class implicit function types. *Proc. ACM Program. Lang.* 4, ICFP (2020), 101:1–101:29. https://doi.org/10.1145/3408983

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308 arXiv:https://academic.oup.com/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf

Pascal Y. Lasnier, Jeremy Yallop, and Magnus O. Myreen. 2026. Brack: A Verified Compiler for Scheme via CakeML. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2026, Rennes, France, January 12-13, 2026*, Kathrin Stark, Yannick Zakowski, Nikhil Swamy, and Nicolas Tabareau (Eds.). ACM, 157–170. https://doi.org/10.1145/3779031.3779098

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/S10817-009-9155-4

Zhaohui Luo. 1989. ECC, an Extended Calculus of Constructions. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 386–395. https://doi.org/10.1109/LICS.1989.39193

Per Martin-Löf. 1984. *Intuitionistic type theory*. Studies in proof theory, Vol. 1. Bibliopolis.

Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 271–283. https://doi.org/10.1145/237721.237791

J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-based typed assembly language. *J. Funct. Program.* 12, 1 (2002), 3–88. https://doi.org/10.1017/S0956796801004178

J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to Typed Assembly Language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 85–97. https://doi.org/10.1145/268946.268954

George C. Necula. 1997. Proof-Carrying Code. In *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*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 106–119. https://doi.org/10.1145/263699.263712

Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional optimizations for CertiCoq. *Proc. ACM Program. Lang.* 5, ICFP, Article 86 (Aug. 2021), 30 pages. https://doi.org/10.1145/3473591

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6 (2019), 125:1–125:36. https://doi.org/10.1145/3280984

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1384)*, Bernhard Steffen (Ed.). Springer, 151–166. https://doi.org/10.1007/BFB0054170

François Pottier and Nadji Gauthier. 2004. Polymorphic typed defunctionalization. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 89–98. https://doi.org/10.1145/964001.964009

Eike Ritter. 1994. Categorical Abstract Machines for Higher-Order Typed lambda-Calculi. *Theor. Comput. Sci.* 136, 1 (1994), 125–162. https://doi.org/10.1016/0304-3975(94)00125-3

The Rocq Development Team. 2025. The Rocq Reference Manual. https://rocq-prover.org/doc/V9.0.0/refman/index.html.

Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. 2002. A type system for certified binaries. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 217–232. https://doi.org/10.1145/503272.503293

Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, François Pottier and George C. Necula (Eds.). ACM, 53–66. https://doi.org/10.1145/1190315.1190324

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *J. Funct. Program.* 29 (2019), e2. https://doi.org/10.1017/S0956796818000229

David Tarditi, J. Gregory Morrisett, Perry Cheng, Christopher A. Stone, Robert Harper, and Peter Lee. 1996. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 181–192. https://doi.org/10.1145/231379.231414

Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 60–76. https://doi.org/10.1145/75277.75283

Stephen Weeks. 2006. Whole-program compilation in MLton. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, Andrew Kennedy and François Pottier (Eds.). ACM, 1. https://doi.org/10.1145/1159876.1159877

Simon Winwood and Manuel M. T. Chakravarty. 2011. Singleton: a general-purpose dependently-typed assembly language. In *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*, Stephanie Weirich and Derek Dreyer (Eds.). ACM, 3–14. https://doi.org/10.1145/1929553.1929557

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/INCO.1994.1093

Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 169–180. https://doi.org/10.1145/507635.507657