

Fully Dependent Abstract Machines (Extended Version)

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 λ -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.

Additional Key Words and Phrases: Keywords

1 Introduction

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

$$\text{app} : \Delta :: (A \rightarrow B) :: A \longrightarrow \Delta :: B$$

whose type indicates that it applies the second-to-top item, a closure of type $A \rightarrow B$, 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 support dependently typed computation, the instruction's type must track the values on the stack and how they change:

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

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

Several recent advances have overcome challenges in preserving dependent types through compilation, adapting 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 *λ -normal form* [Koronkevich et al. 2022]. A key missing requirement is a target language for type-preserving compilers, a low-level language that supports fully dependent types and is a suitable target for translation from the output languages of these transformations. Dependently typed low-level languages date back to Ritter [1994], but 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: a well-typed machine configuration always terminates and produces the correct value according to its type specification (Corollary 2.27). Finally, we define a compilation function from CC_ω to SECD instructions (§2.7) and illustrate that type preservation supports correct separate compilation (Corollary 2.31).

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.

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].

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

Fig. 1. Syntax of CC_ω

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	
TY-UNIVERSE	TY-PI	TY-SIGMA	
$\vdash \Gamma$	$\Gamma \vdash A : U_i \quad \Gamma, x:A \vdash B : U_j$	$\Gamma \vdash A : U_i \quad \Gamma, x:A \vdash B : U_j$	
$\Gamma \vdash U_i : U_{i+1}$	$\Gamma \vdash \Pi x:A. B : U_{\max(i,j)}$	$\Gamma \vdash \Sigma x:A. B : U_{\max(i,j)}$	

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 $A \rightarrow B$ and $A \times B$ for non-dependent functions and pairs (when x is not free in B).

		CC_ω	
TY-LAMBDA	TY-APPLY	TY-PAIR	TY-FST
$\Gamma, x:A \vdash M : B$	$\Gamma \vdash M : \Pi x:A. B \quad \Gamma \vdash N : A$	$\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]$	$\Gamma \vdash M : \Sigma x:A. B$
$\Gamma \vdash \lambda x:A. M : \Pi x:A. B$	$\Gamma \vdash M N : B[N/x]$	$\Gamma \vdash \langle M, N \rangle : \Sigma x:A. B$	$\Gamma \vdash \text{fst } M : A$
			TY-SND
			$\Gamma \vdash M : \Sigma x:A. B$
			$\Gamma \vdash \text{snd } M : B[\text{fst } M/x]$

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.

$$\begin{array}{c} \text{EQ-REDUCE} \\ \frac{\Gamma \vdash M \triangleright^* M' : A \quad \Gamma \vdash N \triangleright^* M' : A}{\Gamma \vdash M \equiv N : A} \end{array} \qquad \begin{array}{c} \text{TY-EQUIV} \\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash A \equiv B : U}{\Gamma \vdash M : B} \end{array}$$

CC_ω

Natural numbers. The natural number type has the usual constructors zero and suc. Its eliminator $\text{rec}_P(M, (x, p).M', N)$ (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 $N = \text{zero}$. When $N = \text{suc } N'$, 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. $\text{rec}_P(M, (x, p).M', N')$. Note that the recursor supports large elimination: we could produce types by recursing over a natural number.

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

CC_ω

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 $\Gamma, x:A \vdash B \equiv B' : U$.
- If $\Gamma \vdash (\Sigma x:A. B) \equiv (\Sigma x:A'. B') : U$, then $\Gamma \vdash A \equiv A' : U$ and $\Gamma, x:A \vdash B \equiv B' : U$.

LEMMA 2.3 (NON-CONFUSION OF TYPE FORMERS). *Different type formers cannot be equivalent. Specifically, $\Gamma \vdash U_i \equiv \text{Nat} : U$ is not derivable, $\Gamma \vdash U_i \equiv \text{Unit} : U$ 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 $\vdash M : (\Pi A : U_0. A)$.*

		CC _ω
$\boxed{\Gamma \vdash M \triangleright N : A}$		(Reduction)
RED-BETA	RED-FST	
$\frac{\Gamma, x:A \vdash N : B \quad \Gamma \vdash M : A}{\Gamma \vdash (\lambda x:A. N) M \triangleright N[M/x] : B[M/x]}$	$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \text{fst } \langle M, N \rangle \triangleright M : A}$	
RED-SND	RED-RECZ	
$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \text{snd } \langle M, N \rangle \triangleright N : B[M/x]}$	$\frac{\Gamma, x:\text{Nat} \vdash P : U \quad \Gamma \vdash M : P[\text{zero}/x] \quad \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{suc } x/x]}{\Gamma \vdash \text{rec}_P(M, (x, p).M', \text{zero}) \triangleright M : P[\text{zero}/x]}$	
RED-RECS		
$\frac{\Gamma, x:\text{Nat} \vdash P : U \quad \Gamma \vdash N : \text{Nat} \quad \Gamma \vdash M : P[\text{zero}/x] \quad \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{suc } x/x]}{\Gamma \vdash \text{rec}_P(M, (x, p).M', \text{suc } N) \triangleright M' [N/x, \text{rec}_P(M, (x, p).M', N)/p] : P[\text{suc } N/x]}$		
(a) Reduction		
$\boxed{\Gamma \vdash M \equiv N : A}$		(Equivalence)
EQ-ETAPI	EQ-ETASIGMA	
$\frac{\Gamma \vdash M : \Pi x:A. B \quad \Gamma \vdash N : \Pi x:A. B \quad \Gamma, x:A \vdash M x \equiv N x : B}{\Gamma \vdash M \equiv N : \Pi x:A. B}$	$\frac{\Gamma \vdash M : \Sigma x:A. B}{\Gamma \vdash \langle \text{fst } M, \text{snd } M \rangle \equiv M : \Sigma x:A. B}$	
(b) Equivalence		

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

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 $\langle \text{env}, \text{ins} \rangle$. The runtime environments and stacks are lists of values. A call frame is a list of triples $\langle \text{ins}, \text{env}, \text{st} \rangle$ 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 $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \longrightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle$. Most instructions operate straightforwardly within the current call frame,

				SECD
Instructions	l	$::=$	$\text{var } x \mid \text{st } n \mid \text{ty } A \mid \text{pushc ins}$ $\mid \text{app} \mid \text{clo } n \text{ ins} \mid \text{pair} \mid \text{fst} \mid \text{snd}$ $\mid \text{unit} \mid \text{lit } n \mid \text{suc} \mid \text{rec } P \text{ ins ins}'$	
Instruction sequences	ins	$::=$	$\text{ret} \mid l; \text{ins}$	
Values	v	$::=$	$n \mid \langle \rangle \mid A \mid \langle v, v' \rangle \mid \langle \text{env}, \text{ins} \rangle$	
Runtime env./stack	env, st	$::=$	$\cdot \mid \text{env} :: v$	
Call frames	fr	$::=$	$\cdot \mid \text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle$	
Machine configuration	C	$::=$	$\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$	

Fig. 3. Syntax of the SECD machine

e.g. $\text{var } x$ looks up the value at position x in the runtime environment¹ and copies it to the stack, $\text{ty } A$ pushes a CC_ω type onto the stack and closes it with the current environment, etc.

Evaluating function calls. Closures $\langle \text{env}, \text{ins} \rangle$ are created by pushc ins that pairs the instruction with the current environment (**SECD-RED-PUSHC**) or by $\text{clo } n \text{ ins}$ 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 $\langle \text{env}', \text{ins}' \rangle$ and a value v , then the machine saves the caller frame's 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**).

Evaluating recursor. The recursor instruction $\text{rec } P \text{ ins}_z \text{ ins}_s$ 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.

2.3 Typing judgements for Instructions

We now assign dependent types to instructions of the SECD machine, using terms in CC_ω to model the stack's contents abstractly. The typing judgement of instructions (and sequences) has the form $\Gamma \vdash \text{ins} : \sigma \rightarrow \sigma'$, where σ is a list of well-typed CC_ω 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 $\Gamma \vdash \sigma \equiv \sigma'$.

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

SECD				
Rule	Instruction	Env.	Stack	Frame
R	ins ins'	env env'	st st'	fr fr'
means that $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \longrightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle$ for rule SECD-RED-R.				
VAR	var x ; ins ins	env env	st st :: env(x)	fr fr
ST	st n ; ins ins	env env	st st :: st(n)	fr fr
TY	ty A ; ins ins	env env	st st :: $A[\gamma]$	fr fr
given a proof that env implements a closing substitution γ .				
PUSHC	pushc ins ; ins' ins'	env env	st st :: $\langle \text{env}, \text{ins} \rangle$	fr fr
APP	app ; ins ins'	env env' :: v	st :: $\langle \text{env}', \text{ins}' \rangle$:: v ·	fr fr :: $\langle \text{ins}, \text{env}, \text{st} \rangle$
CLO	clo n ins ; ins' ins'	env env	st :: $v_1, \dots, v_n$ st :: $\langle v_1 :: \dots :: v_n, \text{ins} \rangle$	fr fr
PAIR	pair ; ins ins	env env	st :: v :: v' st :: $\langle v, v' \rangle$	fr fr
FST	fst ; ins ins	env env	st :: $\langle v, v' \rangle$ st :: v	fr fr
SND	snd ; ins ins	env env	st :: $\langle v, v' \rangle$ st :: v'	fr fr
UNIT	unit ; ins ins	env env	st st :: $\langle \rangle$	fr fr
LIT	lit n ; ins ins	env env	st st :: n	fr fr
SUC	suc ; ins ins	env env	st :: n st :: $(n + 1)$	fr fr
RECZ	rec P ins _z ins _s ; ins ins _z	env env	st :: 0 ·	fr fr :: $\langle \text{ins}, \text{env}, \text{st} \rangle$
RECS	rec P ins _z ins _s ; ins rec P ins _z ins _s ; app ; ret	env env	st :: $(n + 1)$ $\langle \text{env} :: n, \text{ins}_s \rangle$:: n	fr fr :: $\langle \text{ins}, \text{env}, \text{st} \rangle$
RET	ret ins'	env env'	st :: v st' :: v	fr :: $\langle \text{ins}', \text{env}', \text{st}' \rangle$ fr

Fig. 4. Operational semantics of dependent SECD

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

SECD-TY-VAR

$$\frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash \text{var } x : \sigma \rightarrow \sigma :: x}$$

SECD-TY-TY

$$\frac{\Gamma \vdash A : U}{\Gamma \vdash \text{ty } A : \sigma \rightarrow \sigma :: A}$$

SECD

Instruction `pushc 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, `clo n 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-TY-PUSHC

$$\frac{\Gamma, x:A \vdash \text{ins} : \cdot \rightarrow \sigma' :: M}{\Gamma \vdash \text{pushc ins} : \sigma \rightarrow \sigma :: \lambda x:A. M}$$

SECD-TY-CLO

$$\frac{\Gamma \vdash \delta : \Delta \quad |\Delta| = n \quad \Delta, x:A \vdash \text{ins} : \cdot \rightarrow \sigma' :: M}{\Gamma \vdash \text{clo } n \text{ ins} : \sigma :: \delta \rightarrow \sigma :: (\lambda x:A. M)[\delta]}$$

SECD

SECD-TY-APP

$$\frac{\Gamma \vdash M : \Pi x:A. B \quad \Gamma \vdash N : A}{\Gamma \vdash \text{app} : \sigma :: M :: N \rightarrow \sigma :: M N}$$

SECD-TY-RET

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{ret} : \sigma :: M \rightarrow \sigma :: M}$$

Typing for the recursor instruction `rec P insz inss` appears complicated but is conceptually simple: given a predicate over natural numbers P , a base case `insz` that produces M , an inductive case `inss` 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 $\text{rec}_P(M, (x, p).M', N)$ ([SECD-TY-REC](#)).

SECD-TY-REC

$$\frac{\begin{array}{l} \Gamma, x:\text{Nat} \vdash P : U \quad \Gamma \vdash N : \text{Nat} \\ \Gamma \vdash \text{ins}_z : \cdot \rightarrow \sigma' :: M \quad \Gamma \vdash M : P[\text{zero}/x] \\ \Gamma, x:\text{Nat}, p:P \vdash \text{ins}_s : \cdot \rightarrow \sigma'' :: M' \quad \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{suc } x/x] \end{array}}{\Gamma \vdash \text{rec } P \text{ ins}_z \text{ ins}_s : \sigma :: N \rightarrow \sigma :: \text{rec}_P(M, (x, p).M', N)}$$

SECD

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		
$\frac{\text{SECD-TY-PAIR} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \text{pair} : \sigma :: M \rightarrow \sigma :: \langle M, N \rangle}$	$\frac{\text{SECD-TY-FST} \quad \Gamma \vdash M : \Sigma x:A. B}{\Gamma \vdash \text{fst} : \sigma :: M \rightarrow \sigma :: \text{fst } M}$	
$\frac{\text{SECD-TY-SND} \quad \Gamma \vdash M : \Sigma x:A. B}{\Gamma \vdash \text{snd} : \sigma :: M \rightarrow \sigma :: \text{snd } M}$	$\frac{\text{SECD-TY-UNIT} \quad \vdash \Gamma}{\Gamma \vdash \text{unit} : \sigma \rightarrow \sigma :: \langle \rangle}$	$\frac{\text{SECD-TY-LIT} \quad \vdash \Gamma}{\Gamma \vdash \text{lit } n : \sigma \rightarrow \sigma :: n}$
$\frac{\text{SECD-TY-ST} \quad \vdash \Gamma \quad n < \sigma \quad \sigma(n) = M}{\Gamma \vdash \text{st } n : \sigma \rightarrow \sigma :: M}$	$\frac{\text{SECD-TY-SEQ} \quad \Gamma \vdash l : \sigma \rightarrow \sigma' \quad \Gamma \vdash \text{ins} : \sigma' \rightarrow \sigma''}{\Gamma \vdash l ; \text{ins} : \sigma \rightarrow \sigma''}$	$\frac{\text{SECD-TY-EQUIV} \quad \Gamma \vdash \sigma_1 \equiv \sigma'_1 \quad \Gamma \vdash \sigma_2 \equiv \sigma'_2 \quad \Gamma \vdash \text{ins} : \sigma_1 \rightarrow \sigma_2}{\Gamma \vdash \text{ins} : \sigma'_1 \rightarrow \sigma'_2}$

2.4 Machine Well-Formedness

Intuitively, a configuration $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$ 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) $\boxed{v \models M : A}$ The runtime value v implements a closed CC_ω term M of type A .
- (2) $\boxed{\text{env} \models \gamma : \Gamma}$ The environment env implements a closing substitution $\cdot \vdash \gamma : \Gamma$ for context Γ .
- (3) $\boxed{(\text{env} ;_\gamma \Gamma) \vdash \text{st} \models \sigma}$ The stack st implements the abstract stack σ , given some env that implements $\cdot \vdash \gamma : \Gamma$.
- (4) $\boxed{(\text{env} ;_\gamma \Gamma) \vdash \text{fr} : [M : A]}$ 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 $(\text{env} ;_\gamma \Gamma)$.

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 $v \models M : A$ and for environments $\text{env} \models \gamma : \Gamma$ are defined mutual-inductively. Natural numbers, the unit value, and closed types implement themselves. A closure $\langle \text{env}, \text{ins} \rangle$ 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 $\langle v, v' \rangle$ implements $\langle M, N \rangle$ ([SECD-VAL-PAIR](#)). An empty environment implements the empty substitution ϵ . If $\text{env} \models \gamma : \Gamma$ and v implements a closed term M of type $A[\gamma]$, then $\text{env} :: v$ implements the extended context $\Gamma, x : A$ with an extended substitution $\gamma, M/x$ ([SECD-ENV-CONS](#)).

Runtime stack. The judgement for runtime stacks $(\text{env} ;_\gamma \Gamma) \vdash \text{st} \models \sigma$ states that, given some env that implements Γ as γ , st implements the abstract stack σ . We check if $(\text{env} ;_\gamma \Gamma)$ is well-typed at the base case. Given st that implements σ , $\text{st} :: v$ 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-VAL-CLO $\frac{\Gamma, x:A \vdash M : B \quad \text{env} \models \gamma : \Gamma \quad \Gamma, x:A \vdash \text{ins} : \cdot \rightarrow \sigma :: M}{\langle \text{env}, \text{ins} \rangle \models (\lambda x:A. M)[\gamma] : (\Pi x:A. B)[\gamma]}$	SECD-VAL-PAIR $\frac{x:A \vdash B : U \quad v \models M : A \quad v' \models N : B[M/x]}{\langle v, v' \rangle \models \langle M, N \rangle : \Sigma x:A. B}$	SECD
SECD-ENV-CONS $\frac{\Gamma \vdash A : U \quad \text{env} \models \gamma : \Gamma \quad v \models M : A[\gamma]}{\text{env} :: v \models \gamma, M/x : \Gamma, x:A}$	SECD-ST-CONS $\frac{(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma \quad v \models M' : A[\gamma] \quad \Gamma \vdash M : A \quad \vdash M[\gamma] \equiv M' : A[\gamma]}{(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} :: v \models \sigma :: M}$	

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 $(\text{env} ;_{\gamma} \Gamma)$ to the caller frame with environment $(\text{env}' ;_{\gamma'} \Gamma')$. 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 $v \models N$ for some $N \equiv M[\gamma] \equiv M'[\gamma']$.

$$\begin{array}{c}
 \text{SECD-FR-CONS} \\
 \Gamma \vdash \text{ins} : \sigma :: M \rightarrow \sigma' :: N \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B \\
 \text{env} \models \gamma : \Gamma \quad (\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma \quad (\text{env}' ;_{\gamma'} \Gamma') \vdash \text{fr} : [N' : B'] \\
 \vdash B[\gamma] \equiv B'[\gamma'] : U \quad \vdash N[\gamma] \equiv N'[\gamma'] : B[\gamma] \\
 \hline
 (\text{env} ;_{\gamma} \Gamma) \vdash \text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle : [M : A]
 \end{array}$$

SECD

The typing judgement $(\text{env} ;_{\gamma} \Gamma) \vdash \text{fr} : [M : A]$ is defined exactly for this purpose. It states that the top frame of fr has $(\text{env} ;_{\gamma} \Gamma)$ 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 $\langle \text{ins}, \text{env}, \text{st} \rangle$ 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 $(\text{env}' ;_{\gamma'} \Gamma')$ 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 $(\text{env} ;_{\gamma} \Gamma)$ 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 $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$ is well-formed, denoted as $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$, if the following conditions hold:*

- (1) $\Gamma \vdash \text{ins} : \sigma \rightarrow \sigma' :: M$ for some Γ, σ, σ' , and M , with $\Gamma \vdash M : A$ for some A .
- (2) $\text{env} \models \gamma : \Gamma$ for some γ .
- (3) $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma$.

- (4) $(\text{env}' :_{\gamma'} \Gamma') \vdash \text{fr} : [M' : A']$, 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 $v \models M : \Pi x:A. B$, 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 $\Gamma \vdash \text{ins} : \sigma \rightarrow \sigma'$, then $\vdash \Gamma$ and every term in (σ, σ') is well-typed under Γ .
- If $\text{env} \models \gamma : \Gamma$, then $\cdot \vdash \gamma : \Gamma$ and $|\text{env}| = |\Gamma|$.
- If $v \models M : A$, then $\vdash M : A$.
- If $(\text{env} :_{\gamma} \Gamma) \vdash \text{st} \models \sigma$, then $\text{env} \models \gamma : \Gamma$, $|\text{st}| = |\sigma|$, and every term in σ is well-typed under Γ .
- If $(\text{env} :_{\gamma} \Gamma) \vdash \text{fr} : [M : A]$, then $\text{env} \models \gamma : \Gamma$ and $\Gamma \vdash M : A$.

LEMMA 2.8 (CANONICITY OF VALUES). *If $v \models M : A$, then*

- If $A \equiv_{nf} U_i$, then $v = A'$ (v is a type).
- If $A \equiv_{nf} \text{Nat}$, then $v = n$ for some natural number n .
- If $A \equiv_{nf} \text{Unit}$, then $v = \langle \rangle$.
- If $A \equiv_{nf} \Pi x:A'. B'$, then $v = \langle \text{env}, \text{ins} \rangle$ for some env, ins .
- If $A \equiv_{nf} \Sigma x:A'. B'$, then $v = \langle v_1, v_2 \rangle$ for some v_1, 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). $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$ *halts with result* v , denoted as $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \Downarrow v$, if $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \rightarrow^* \langle \text{ret}, \text{env}', \text{st}' :: v, \cdot \rangle$.

$\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$ *halts*, denoted as $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \Downarrow$, if there exists v such that $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \Downarrow v$.

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).

We state a few useful lemmas of the dependent SECD machine, all of which can be proved easily by induction. Lemma 2.10 and Lemma 2.11 state that we obtain well-typed values from well-formed environments and stacks. In particular, given an environment $(\text{env} :_{\gamma} \Gamma)$, the value at position x of the environment implements $x[\gamma]$; given a stack $\text{st} \models \sigma$ under this environment, and the value at position n of the stack implements $M[\gamma]$, where M is the n -th term of the abstract stack σ .

LEMMA 2.10 (ENVIRONMENT ACCESS). *If $\text{env} \models \gamma : \Gamma$ and $\Gamma \vdash x : A$, then there exists v such that $\text{env}(x) = v$ and $v \models x[\gamma] : A[\gamma]$.*

LEMMA 2.11 (STACK ACCESS). *If $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma$ and $n < |\text{st}|$, then there exists v such that $\text{st}(n) = v$ and $v \models M[\gamma] : A[\gamma]$, where $M = \sigma(n)$ and $\Gamma \vdash M : A$.*

Syntactically, stacks and environments are both lists of runtime values. Lemma 2.12 states that if a stack implements an abstract stack σ under $(\text{env} ;_{\gamma} \Gamma)$ and σ can be viewed as a substitution, then the stack can be viewed as an environment that implements the composed substitution $\sigma \circ \gamma$. We can also split a well-typed stack into two (Lemma 2.13).

LEMMA 2.12 (STACK AS ENVIRONMENT). *If $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma$ and $\Gamma \vdash \sigma : \Delta$, then $\text{st} \models \sigma \circ \gamma : \Delta$.*

LEMMA 2.13 (STACK SPLIT). *If $(\text{env} ;_{\gamma} \Gamma) \vdash (\text{st}' :: \text{st}) \models (\sigma :: \delta)$ and $|\text{st}| = |\delta|$, then $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \delta$ and $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st}' \models \sigma$.*

2.5.1 Progress.

THEOREM 2.14 (PROGRESS). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$, then either $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \longrightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle$ or $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$ halts.*

PROOF. By induction on the instruction sequence ins . We first consider the cases that $\text{ins} = l ; \text{ins}$, which can be grouped as follows:

- (1) The machine steps unconditionally: $l = \text{ty}, \text{pushc}, \text{unit}, \text{lit}$.
- (2) The machine steps if a value could be found from the environment or the stack: $l = \text{var}, \text{st}$.
- (3) The machine steps if there are enough values on top of the stack: $l = \text{clo}, \text{pair}$.
- (4) The machine steps if there are enough values of a certain shape on top of the stack: $l = \text{app}, \text{fst}, \text{snd}, \text{suc}, \text{recZ}, \text{recS}$.

Case (1) is trivial. Case (2) is implied by the environment and stack access lemmas. Case (3) is proved by well-formedness of judgements (Lemma 2.7), which guarantees that enough values exist on the stack. Case (4) requires both Lemma 2.7 and canonicity of values (Lemma 2.8) to show that the stack has enough values of the desired shape on the top.

We sketch the proofs for clo and app . Other cases follow similarly.

Case clo . By assumption, $\Gamma \vdash \text{clo } n \text{ ins} : \sigma :: \delta \rightarrow \sigma :: (\lambda x:A. M)[\delta]$ with $|\delta| = n$. Since the current stack st implements the abstract stack $\sigma :: \delta$, by Lemma 2.7, their size must be equal, and st could be split into $\text{st} = \text{st}' :: \text{st}''$ with $|\text{st}''| = n$, and the machine steps by **SECD-RED-CLO**.

Case app . By assumption, $\Gamma \vdash \text{app} : \sigma :: M :: N \rightarrow \sigma :: M N$ with $\Gamma \vdash M : \Pi x:A. B$ and $\Gamma \vdash N : A$, and $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models (\sigma :: M :: N)$. So, the current stack must be in the form of $\text{st} = \text{st}' :: v :: v'$ with $v \models M' : (\Pi x:A. B)[\gamma]$ and $v' \models N' : A[\gamma]$. Since $(\Pi x:A. B)[\gamma] = \Pi x:(A[\gamma]). B[\gamma]$, we know that v is a value of the function type. By Lemma 2.8, $v = \langle \text{env}, \text{ins}' \rangle$ and the machine could step by **SECD-RED-APP**.

The final case is that $\text{ins} = \text{ret}$.

Case ret . By assumption, $\Gamma \vdash \text{ret} : \sigma :: M \rightarrow \sigma :: M$. By Lemma 2.7, the current stack must be in the shape of $\text{st} = \text{st}' :: v$, i.e. there is a return value ready on the stack. If the call-frame stack is empty, then the machine halts. Otherwise, the machine restores from the top call frame and steps by **SECD-RED-RET**.

□

2.5.2 Preservation.

THEOREM 2.15 (PRESERVATION). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$ and $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \longrightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle$, then $\vdash \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle$.*

PROOF. By induction on the instruction sequence ins . We first consider the cases that $\text{ins} = \text{!}; \text{ins}$, which can be grouped as follows:

- (1) The instruction pushes a value on the stack: $\text{!} = \text{var}, \text{st}, \text{ty}, \text{pushc}, \text{unit}, \text{lit}$.
- (2) The instruction modifies part of the stack: $\text{!} = \text{clo}, \text{pair}, \text{fst}, \text{snd}, \text{suc}$.
- (3) The function application: $\text{!} = \text{app}$.
- (4) The recursor: $\text{!} = \text{rec}$, with two sub-cases **SECD-RED-RECZ** and **SECD-RED-RECS**.

Case (1) requires the value pushed to be well-typed, which is straightforward by assumption using the environment/stack access lemmas or the typing judgements for values. Case (2) is proved similarly, showing that the modified stack contains well-typed values by assumption. Cases (3) and (4) involves allocating a new call frame, which requires showing that the new frame is compatible with the previous one.

We give detailed proofs for the cases of var , clo , app , and the successor case for rec (one from each group). Other cases follow similarly.

Case var. By assumption, $\Gamma \vdash \text{var } x : \sigma \rightarrow \sigma :: x$ with $\Gamma \vdash x : A$. The machine steps by **SECD-RED-VAR**, pushing a value $v = \text{env}(x)$ on the stack. The new configuration is well-formed if we can show that $(\text{env} ;_{\gamma} \Gamma) \vdash (\text{st} :: v) \models (\sigma :: x)$. We have $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma$ by assumption, and $v \models x[\gamma] : A[\gamma]$ by Lemma 2.10. So, we have the goal by **SECD-ST-CONS**.

Case clo. By assumption, $\Gamma \vdash \text{clo } n \text{ ins} : \sigma :: \delta \rightarrow \sigma :: (\lambda x:A.M)[\delta]$ with $\Gamma \vdash \delta : \Delta$, $|\Delta| = n$, and $\Delta, x:A \vdash \text{ins} : \cdot \rightarrow \sigma' :: M$. The machine steps by **SECD-RED-CLO**, splitting the current stack into $\text{st}' :: \text{st}$ and pushing a closure $\langle \text{st}, \text{ins} \rangle$ on st' . The new configuration is well-formed if the new stack implements $\sigma :: (\lambda x:A.M)[\delta]$. Since the current stack implements $\sigma :: \delta$ and $|\Delta| = |\text{st}| = n$, by Lemma 2.13, we have $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st}' \models \sigma$ and $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \delta$. So, st' implements σ . By Lemma 2.12, $\text{st} \models \delta \circ \gamma : \Gamma$. So, $\langle \text{st}, \text{ins} \rangle \models (\lambda x:A.M)[\delta][\gamma] : (\Pi x:A.B)[\delta][\gamma]$ by **SECD-VAL-CLO**, and the new stack is well-formed by **SECD-ST-CONS**.

Case app. The current stack has a closure $\langle \text{env}', \text{ins}' \rangle$ and a value v on the top. The machine steps by **SECD-RED-APP**:

$$\langle \text{app} ; \text{ins}, \text{env}, \text{st} :: \langle \text{env}', \text{ins}' \rangle :: v, \text{fr} \rangle \longrightarrow \langle \text{ins}', \text{env}' :: v, \cdot, \text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle \rangle$$

where $\Gamma \vdash \text{app} : \sigma :: M :: N \rightarrow \sigma :: M N$ with $\Gamma \vdash M : \Pi x:A.B$ and $\Gamma \vdash N : A$.

We start by fully elaborating the type information we have from the assumptions. We know that the initial stack $\text{st} :: \langle \text{env}', \text{ins}' \rangle :: v$ implements $\sigma :: M :: N$. By **SECD-ST-CONS**, we must have $\langle \text{env}', \text{ins}' \rangle \models (\lambda x:A'.N')[\delta] : (\Pi x:A'.B')[\delta]$ with $\Delta, x:A' \vdash \text{ins}' : \cdot \rightarrow \sigma'' :: N'$ and $\text{env}' \models \delta : \Delta$ such that $(\Pi x:A.B)[\gamma] \equiv (\Pi x:A'.B')[\delta]$ and $M[\gamma] \equiv (\lambda x:A'.N')[\delta]$. By Lemma 2.2, $A[\gamma] \equiv A'[\delta]$ and $x:A \vdash B[\gamma] \equiv B'[\delta] : U$. We also know that $v \models N[\gamma] : A[\gamma]$. There are six sub-goals to show that the new state is well-formed, according to Definition 2.6:

- (1) The new instruction ins' is well-typed. Indeed, we have $\Delta, x:A \vdash \text{ins}' : \cdot \rightarrow \sigma'' :: N'$.
- (2) The environment $\text{env}' :: v$ implements the instruction's context $\Delta, x:A$. We already have $\text{env}' \models \delta : \Delta$. Since $v \models N[\gamma] : A[\gamma]$ and $A[\gamma] \equiv A'[\delta]$, we have $v \models N[\gamma] : A'[\delta]$, and $\text{env}' :: v \models (\delta, N[\gamma]/x) : (\Delta, x:A)$ by **SECD-ENV-CONS**.
- (3) The empty stack implements an empty stack, trivial.
- (4) The extended frame is well-formed. We can easily show this by **DA-FRAME-CONS** (every premise follows straightforwardly from assumption).

- (5) The return type of ins' matches with the expected return type of ins , i.e. $(B[N/x])[Y] \equiv B'[\delta, N[Y]/x]$. Indeed, $(B[N/x])[Y] \equiv B[Y, N[Y]/x] \equiv B'[\delta, N[Y]/x]$ since $x:A \vdash B[Y] \equiv B'[\delta] : U$.
- (6) The return value of ins' matches with the expected return value of ins , i.e. $(M N)[Y] \equiv N'[\delta, N[Y]/x]$. Since $M[Y] \equiv (\lambda x:A'. N')[\delta]$, we have $(M N)[Y] \equiv (M[Y]) (N[Y]) \equiv ((\lambda x:A'. N')[\delta]) (N[Y]) \equiv N'[\delta, N[Y]/x]$.

Now we can conclude that the new state is well-formed.

Case rec. If the machine steps by **SECD-RED-REC**:

$$\langle \text{rec } P \text{ ins}_z \text{ ins}_s ; \text{ins}, \text{env}, \text{st} :: n, \text{fr} \rangle \longrightarrow \langle \text{rec } P \text{ ins}_z \text{ ins}_s ; \text{app} ; \text{ret}, \text{env}, \cdot :: \langle \text{env} :: n', \text{ins}_s \rangle :: n', \text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle \rangle$$

where $n = n' + 1$, $\Gamma \vdash \text{ins}_z : \cdot \rightarrow \sigma_1 :: M$, $\Gamma, x:\text{Nat}, p:P \vdash \text{ins}_s : \cdot \rightarrow \sigma_2 :: M'$, $\Gamma \vdash N : \text{Nat}$, and $\Gamma \vdash \text{ins} : \sigma :: \text{rec}_P(M, (x, p).M', N) \rightarrow \sigma' :: N'$. We also know that $n \models N[Y] : \text{Nat}$ from assumption, which implies that $n \equiv N[Y]$.

Similar to the application case, we have six sub-goals to verify.

- (1) The new instruction is well-typed. Indeed, we can derive that

$$\Gamma \vdash \text{rec } P \text{ ins}_z \text{ ins}_s ; \text{app} ; \text{ret} : \cdot :: (\lambda p:P.M')[n'/x] :: n' \rightarrow \cdot :: M'[n'/x, L/p]$$

where $L = \text{rec}_P(M, (x, p').M, n')$ by **SECD-TY-REC**, **SECD-TY-APP**, and finally **SECD-TY-RET**.

- (2) The environment env implements Γ , trivial by assumption.
- (3) The stack implements $(\lambda p:P.M')[n'/x] :: n'$. Indeed, $\langle \text{env} :: n', \text{ins}_s \rangle \models (\lambda p:P.M')[n'/x][Y]$ since $\langle \text{env} :: n', \text{ins}_s \rangle \models (\lambda p:P.M')[Y, n'/x]$ (by **SECD-VAL-CLO**) and $(\lambda p:P.M')[Y, n'/x] \equiv (\lambda p:P.M')[n'/x][Y]$.
- (4) The extended frame is well-formed. We can easily show this by **DA-FRAME-CONS** (every premise follows straightforwardly from assumption). Also, n' implements n' .
- (5) The return type of ins_s matches with the expected input type of ins match, i.e. $(P[N/x])[Y] \equiv (P[\text{succ } n'/x])[Y]$. Since $n \equiv N[Y]$ and $n = n' + 1$, we have $(P[N/x])[Y] \equiv P[Y, N[Y]/x] \equiv P[Y, \text{succ } n'/x] \equiv (P[\text{succ } n'/x])[Y]$.
- (6) The return value of ins_s matches with the expected input value of ins match, i.e. $(\text{rec}_P(M, (x, p).M', N))[Y] \equiv (M'[n'/x, \text{rec}_P(M, (x, p').M, n')/p])[Y]$. Using a similar reasoning to (5), we have

$$\begin{aligned} & (\text{rec}_P(M, (x, p).M', N))[Y] \\ & \equiv \text{rec}_{(P[Y])}((M[Y]), (x, p).(M'[Y]), (N[Y])) \\ & \equiv \text{rec}_{(P[Y])}((M[Y]), (x, p).(M'[Y]), (\text{succ } n')) \\ & \equiv M'[Y, n'/x, \text{rec}_{(P[Y])}((M[Y]), (x, p).(M'[Y]), n')/p] \\ & \equiv (M'[n'/x, \text{rec}_P(M, (x, p').M, n')/p])[Y]. \end{aligned}$$

Now we can conclude that the new state is well-formed.

The final case is that $\text{ins} = \text{ret}$.

Case ret. The machine steps by **SECD-RED-RET**:

$$\langle \text{ret}, \text{env}, \text{st} :: v, \text{fr} :: \langle \text{ins}', \text{env}', \text{st}' \rangle \rangle \longrightarrow \langle \text{ins}', \text{env}', \text{st}' :: v, \text{fr} \rangle.$$

By assumption, $\Gamma \vdash \text{ret} : \sigma :: M \rightarrow \sigma :: M$ for some $\Gamma \vdash M : A$. The top call frame is well typed, so we have $\Gamma' \vdash \text{ins}' : \sigma' :: M' \rightarrow \sigma'' :: N'$ for some $\Gamma' \vdash M' : A'$, $\text{env}' \models \gamma' : \Gamma'$, and st' implements σ' . Since the machine is well-formed, we know that $A[Y] \equiv A'[\gamma']$ and $M[Y] \equiv M'[\gamma']$. The new configuration is well-typed if the return value v implements M' ,

in other words, we need to show that $v \models N : A[\gamma]$ where $N \equiv M'[\gamma']$. Indeed, since the current stack is well-typed, we have $v \models N : A[\gamma]$ with $N \equiv M[\gamma] \equiv M'[\gamma']$.

□

2.5.3 Type safety and partial correctness. 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 2.16 (TYPE SAFETY). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$, then it never reduces to a configuration that is irreducible and not at halt.*

Let us examine a well-formed configuration $\langle \text{ret}, \text{env}, \text{st} :: v, \cdot \rangle$ at halt. The type of the return is $\Gamma \vdash \text{ret} : \sigma :: M \rightarrow \sigma :: M$, 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 2.17 (PARTIAL CORRECTNESS). *If $\Gamma \vdash \text{ins} : \sigma \rightarrow \sigma' :: M$ with $\Gamma \vdash M : A$, $\text{env} \models \gamma : \Gamma$, $(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma$, and $\langle \text{ins}, \text{env}, \text{st}, \cdot \rangle \Downarrow v$, then $v \models M[\gamma] : A[\gamma]$.*

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 2.19) and show that a well-formed configuration always terminates at its current frame (Lemma 2.25) by proving a fundamental lemma for the logical relation (Lemma 2.23). Termination then follows from an induction over the size of the call frame stack (Theorem 2.26).

DEFINITION 2.18 (HALT AT CURRENT FRAME). $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$ *halts at its current frame with return value v , denoted as $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \Downarrow v$, if $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \longrightarrow^* \langle \text{ret}, \text{env}, \text{st}' :: v, \text{fr} \rangle$ and the reduction sequence never returns beyond fr .*

$\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$ *halts at its current frame, denoted as $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \Downarrow$, if there exists v such that $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \Downarrow v$.*

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
NF-NAT	NF-PI	NF-SIGMA	
$\frac{A \equiv_{nf} \text{Nat}}{\Vdash A}$	$\frac{A \equiv_{nf} \Pi x:A'. B' \quad \Vdash A' \quad \forall (a:A'). \Vdash B'[a/x]}{\Vdash A}$	$\frac{A \equiv_{nf} \Sigma x:A'. B' \quad \Vdash A' \quad \forall (a:A'). \Vdash B'[a/x]}{\Vdash A}$	

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**), $v \in H(A)$ if v is a well-typed pair $\langle v_1, v_2 \rangle$ and its two components are in the relations of the first/second projection. Specifically, if v implements some closed term M , then we need $v_1 \in H(A')$ and $v_2 \in H(B'[\text{fst } M/x])$ (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 $A \equiv \Pi x:A'. B'$, $v \in H(A)$ if v is a well-typed closure $\langle \text{env}, \text{ins} \rangle$, and for all $v' \in H(A')$ with $v' \models a : A'$, evaluating the closure's code in $\text{env} :: v'$ 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 2.19 (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**: $A \equiv_{nf} \text{Nat}$ $v \in H(A)$ iff $v = n$.
- **NF-UNIT**: $A \equiv_{nf} \text{Unit}$ $v \in H(A)$ iff $v = \langle \rangle$.
- **NF-UNIVERSE**: $A \equiv_{nf} U_i$ $v \in H(A)$ iff $v \models A' : U_i$.
- **NF-PI**: $A \equiv_{nf} \Pi x:A'. B'$ $v \in H(A)$ iff $v = \langle \text{env}, \text{ins} \rangle$, $v \models M : \Pi x:A'. B'$, and for all $v' \in H(A')$ such that $v' \models a : A'$, $\langle \text{ins}, \text{env} :: v', \cdot, \text{fr} \rangle \downarrow v''$ and $v'' \in H(B'[a/x])$.
- **NF-SIGMA**: $A \equiv_{nf} \Sigma x:A'. B'$ $v \in H(A)$ iff $v = \langle v_1, v_2 \rangle$, $v \models M : \Sigma x:A'. B'$, $v_1 \in H(A')$, $v_2 \in H(B'[\text{fst } M/x])$.

LEMMA 2.20 (WELL-TYPED HALTING VALUES). *If $v \in H(A)$, then $v \models M : A$ for some M .*

The halting relations for well-typed environments $\text{env} \Vdash_h \gamma : \Gamma$ and stacks $(\text{env} :_\gamma \Gamma) \vdash \text{st} \Vdash_h \sigma$ 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 their corresponding types.

DEFINITION 2.21 (THE HALTING RELATION ON ENVIRONMENTS). *Let $\text{env} \Vdash_h \gamma : \Gamma$ be a subset of well-typed runtime environments $\text{env} \models \gamma : \Gamma$, defined on the typing derivation of $\text{env} \models \gamma : \Gamma$ recursively:*

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

DEFINITION 2.22 (THE HALTING RELATION ON STACKS). *Let $(\text{env} :_\gamma \Gamma) \vdash \text{st} \Vdash_h \sigma$ be a subset of well-typed runtime stacks $(\text{env} :_\gamma \Gamma) \vdash \text{st} \models \sigma$, defined on the typing derivation of $\text{env} \models \gamma : \Gamma$ recursively:*

- **SECD-ST-NIL** $(\text{env} :_\gamma \Gamma) \vdash \cdot \Vdash_h \cdot$ holds if $\text{env} \models \gamma : \Gamma$.

- SECD-ST-CONS $(\text{env} :_{\gamma} \Gamma) \vdash (\text{st} :: v) \Vdash_h (\sigma :: M)$ if $(\text{env} :_{\gamma} \Gamma) \vdash \text{st} \Vdash_h \sigma$ and $v \in H(A[\gamma])$ (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 $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$ 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 2.23 (FUNDAMENTAL LEMMA). *For all $(\text{ins}, \text{env}, \text{st})$, if $\Gamma \vdash \text{ins} : \sigma \rightarrow \sigma' :: M$ with $\Gamma \vdash M : A$, $\text{env} \Vdash_h \gamma : \Gamma$, and $(\text{env} :_{\gamma} \Gamma) \vdash \text{st} \Vdash_h \sigma$, then $\forall \text{fr}. \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \downarrow v$ and $v \in H(A[\gamma])$.*

PROOF. By induction on the instruction sequence ins . The case $\text{ins} = \text{ret}$ is trivial. We only have to consider $\text{ins} = \text{!}; \text{ins}$, which can be grouped as follows:

- (1) Instructions that do not create a new call stack: $\text{!} = \text{ty}$, pushc , and so on.
- (2) The function application: $\text{!} = \text{app}$.
- (3) The recursor: $\text{!} = \text{rec}$.

For case (1), the machine steps into a next state $\langle \text{ins}, \text{env}, \text{st}', \text{fr} \rangle$ with some new stack st' . We need to show that the new stack is still in Halt, then we apply the induction hypothesis. We show case (2) by definition of the halting relation and case (3) by induction on the natural number n on top of the stack.

We sketch the proofs for pushc , app and rec . Other cases follow similarly.

Case pushc. By assumption, $\text{!} = \text{pushc } \text{ins} ; \text{ins}'$, $\Gamma, x:A \vdash \text{ins} : \cdot \rightarrow \sigma'' :: M$ with $\Gamma, x:A \vdash M : B$, and $\Gamma \vdash \text{ins}' : \sigma :: \lambda x:A. M \rightarrow \sigma' :: N$. The machine steps from $\langle \text{pushc } \text{ins} ; \text{ins}', \text{env}, \text{st}, \text{fr} \rangle$ to $\langle \text{ins}', \text{env}, \text{st} :: \langle \text{env}, \text{ins} \rangle, \text{fr} \rangle$. We aim for applying the induction hypothesis on ins' to show the proof goal. We already have $\text{env} \Vdash_h \gamma : \Gamma$ and $(\text{env} :_{\gamma} \Gamma) \vdash \text{st} \Vdash_h \sigma$ from the assumption, and we need $\langle \text{env}, \text{ins} \rangle \in H((\Pi x:A. B)[\gamma])$ to show that the new stack is in halt.

We can show $\langle \text{env}, \text{ins} \rangle \in H((\Pi x:A. B)[\gamma])$ with the Π -case of the halting relation since $(\Pi x:A. B)[\gamma] = \Pi x:(A[\gamma]). B[\gamma]$. The closure is clearly well-typed. Given some $v' \in H(A[\gamma])$ such that $v \models a : A[\gamma]$, we need to show that $\langle \text{ins}, \text{env} :: v, \cdot, \text{fr} \rangle \downarrow v'$ with $v' \in H(B[a/x])$, which follows directly from the I.H. on ins .

Case app. By assumption, $\text{!} = \text{app} ; \text{ins}$, $\Gamma \vdash \text{ins} : \sigma :: M \rightarrow \sigma' :: N'$ with $\Gamma \vdash M : \Pi x:A. B$ and $\Gamma \vdash N : A$. We also know that $\langle \text{env}', \text{ins}' \rangle \models M[\gamma] : (\Pi x:A. B)[\gamma]$ and $v \models N[\gamma] : A[\gamma]$. The proof goal is to show that $\langle \text{app} ; \text{ins}, \text{env}, \text{st} :: \langle \text{env}', \text{ins}' \rangle :: v, \text{fr} \rangle \downarrow v'$ for some v' .

First, $\langle \text{app} ; \text{ins}, \text{env}, \text{st} :: \langle \text{env}', \text{ins}' \rangle :: v, \text{fr} \rangle$ evaluates to $\langle \text{ins}', \text{env}' :: v, \cdot, \text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle \rangle$ by SECD-RED-APP. By assumption, $\langle \text{env}', \text{ins}' \rangle \in H((\Pi x:A. B)[\gamma])$ and $v \in H(A[\gamma])$, then by the Π -case of Definition 2.19, the machine terminates at frame $\text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle$ with a return value $v' \in H(B[\gamma, N[\gamma]/x])$. Next, the machine returns v' and transits to $\langle \text{ins}, \text{env}, \text{st} :: v', \text{fr} \rangle$, which will terminate and produce a halting value by I.H. since v' is in $H(B[\gamma, N[\gamma]/x])$.

Case rec. By assumption, $\text{!} = \text{rec } P \text{ ins}_z \text{ ins}_s ; \text{ins}$, $\Gamma \vdash \text{ins} : \sigma :: \text{rec}_P(M, (x, p).M', N) \rightarrow \sigma' :: N'$ with $\Gamma \vdash N : \text{Nat}$ and $\Gamma \vdash N' : A$. We first prove the following statement by induction on n .

Proposition: For all $(n, \text{ins}', \text{env}', \text{st}', \text{fr}')$, if $\Gamma' \vdash \text{ins}' : \sigma_1 :: \text{rec}_P(M, (x, p).M', N) \rightarrow \sigma_2 :: N''$ with $\Gamma' \vdash N'' : A$, $\text{env}' \Vdash_h \gamma' : \Gamma'$, $(\text{env}' :_{\gamma'} \Gamma') \vdash \text{st}_1 \Vdash_h \sigma_1$, and $n \models N[\gamma'] : \text{Nat}$, if $\forall v' \in H(P[\gamma', n/x]). \langle \text{ins}', \text{env}', \text{st}' :: v', \text{fr}' \rangle \downarrow v$ for some v and $v \in H(A[\gamma'])$, then $\langle \text{rec } P \text{ ins}_z \text{ ins}_s ; \text{ins}', \text{env}', \text{st}' :: n, \text{fr}' \rangle \downarrow v$ for some v and $v \in H(A[\gamma'])$.

Fix some $\text{ins}', \text{env}', \text{st}', \text{fr}'$ with the desired property. When $n = 0$, we have

$$\begin{aligned} & \langle \text{rec } P \text{ ins}_z \text{ ins}_s ; \text{ins}', \text{env}', \text{st}' :: 0, \text{fr}' \rangle \\ & \longrightarrow \langle \text{ins}_z, \text{env}', \cdot, \text{fr}' :: \langle \text{ins}', \text{env}', \text{st}' \rangle \rangle \quad \text{by } \text{SECD-RED-RECZ} \\ & \longrightarrow^* \langle \text{ret}, \text{env}', \text{st}'' :: v, \text{fr}' :: \langle \text{ins}', \text{env}', \text{st}' \rangle \rangle \quad \text{by I.H. on } \text{ins}_z \text{ with } v \in H(P[\gamma, 0/x]) \\ & \longrightarrow \langle \text{ins}', \text{env}', \text{st}' :: v, \text{fr}' \rangle \quad \text{by } \text{SECD-RED-RET}, \end{aligned}$$

which will halt at the current frame with a halting value by assumption, since $v \in H(P[\gamma, 0/x])$.

When $n = n' + 1$, we have $\langle \text{rec } P \text{ ins}_z \text{ ins}_s ; \text{ins}', \text{env}', \text{st}' :: n, \text{fr}' \rangle$

$\longrightarrow \langle \text{rec } P \text{ ins}_z \text{ ins}_s ; \text{app} ; \text{ret}, \text{env}', \cdot :: \langle \text{env} :: n', \text{ins}_s \rangle :: n', \text{fr}' :: \langle \text{ins}', \text{env}', \text{st}' \rangle \rangle$, which halts at current frame with a halting value by I.H. of the proposition on n . We can apply I.H. because $\langle \text{app} ; \text{ret}, \text{env}, \cdot :: \langle \text{env} :: n', \text{ins}_s \rangle :: v', \text{fr}' :: \langle \text{ins}', \text{env}', \text{st}' \rangle \rangle \downarrow v$ with v in halt for all $v' \in H(P[\gamma, n/x])$. This is true via an argument similar to the proof of the app case: we show that $\langle \text{env} :: n', \text{ins}_s \rangle \in H(\Pi p : (P[\gamma, n/x]). P[\gamma, (\text{succ } n)/x])$ by I.H. on ins_s , then, we derive that applying the halting closure with a halting value will halt at the current frame with a halting result.

Now, by I.H. on ins , $\forall v' \in H(P[\gamma, n/x]). \langle \text{ins}, \text{env}, \text{st} :: v', \text{fr} \rangle \downarrow v$ and $v \in H(A[\gamma])$. Therefore, we can initialize the proposition with $(n, \text{ins}, \text{env}, \text{st}, \text{fr})$ and conclude that $\langle \text{rec } P \text{ ins}_z \text{ ins}_s ; \text{ins}, \text{env}, \text{st} :: n, \text{fr} \rangle \downarrow v$ with $v \in H(A[\gamma])$.

□

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 2.24). Then, given $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$, since the environment and the stack are well-typed, they are in the halting relation and we have $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \downarrow v$ (Lemma 2.25). 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 2.26).

LEMMA 2.24 (WELL-TYPED VALUES HALT). *Well-typed values, environments, and stacks are in their respective halting relations.*

- If $v \models a : A$, then $v \in H(A)$.
- If $\text{env} \models \gamma : \Gamma$, then $\text{env} \models_h \gamma : \Gamma$.
- If $(\text{env} :_\gamma \Gamma) \vdash \text{st} \models \sigma$, then $(\text{env} :_\gamma \Gamma) \vdash \text{st} \models_h \sigma$.

LEMMA 2.25 (TERMINATION AT CURRENT FRAME). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$, then $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \downarrow v$ and $v \in H(A)$ where $v \models M : A$.*

THEOREM 2.26 (TERMINATION). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$, then $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \Downarrow v$ for some v .*

PROOF. By induction on fr . If there is no call frame, then $\langle \text{ins}, \text{env}, \text{st}, \cdot \rangle \downarrow v$ by Lemma 2.25, and therefore $\langle \text{ins}, \text{env}, \text{st}, \cdot \rangle \Downarrow v$ by definition. If $\text{fr} = \text{fr} :: \langle \text{ins}', \text{env}', \text{st}' \rangle$, then the machine steps into $\langle \text{ins}', \text{env}', \text{st}' :: v, \text{fr} \rangle$ for some v since $\langle \text{ins}, \text{env}, \text{st}, \text{fr} :: \langle \text{ins}', \text{env}', \text{st}' \rangle \rangle \downarrow v$ (by Lemma 2.25), and by I.H. on fr , $\langle \text{ins}', \text{env}', \text{st}' :: v, \text{fr} \rangle \Downarrow v'$. So, $\langle \text{ins}, \text{env}, \text{st}, \text{fr} :: \langle \text{ins}', \text{env}', \text{st}' \rangle \rangle \Downarrow v'$. □

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 2.27). The corresponding result for complete programs (i.e. closed instruction sequences with empty initial stacks) follows as a corollary (Corollary 2.28).

COROLLARY 2.27 (TOTAL CORRECTNESS). *If $\Gamma \vdash \text{ins} : \sigma \rightarrow \sigma' :: M$ with $\Gamma \vdash M : A$, $\text{env} \models \gamma : \Gamma$, and $(\text{env} \cdot_\gamma \Gamma) \vdash \text{st} \models \sigma$, then $\langle \text{ins}, \text{env}, \text{st}, \cdot \rangle \Downarrow v$ with $v \models M[\gamma] : A[\gamma]$.*

COROLLARY 2.28 (TOTAL CORRECTNESS FOR PROGRAMS). *If $\cdot \vdash \text{ins} : \cdot \rightarrow \sigma :: M$ with $\vdash M : A$, then $\langle \text{ins}, \cdot, \cdot, \cdot \rangle \Downarrow v$ with $v \models M : A$.*

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 $\llbracket - \rrbracket_c$ 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 $\llbracket - \rrbracket_v$ in Fig. 5b that transforms closed CC_ω values to SECD values. The compilation scheme is shown type-preserving by a simple induction (Lemma 2.29). Therefore, by total correctness, a closed CC_ω term M compiles to an SECD program that terminates and produces a result implementing M (Corollary 2.30). Similarly, given an open CC_ω term M and a list of closed CC_ω values γ , executing $\llbracket M \rrbracket_c$ 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 2.31).

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.

$\llbracket \text{CC}_\omega \rrbracket$		
$\llbracket x \rrbracket = \text{var } x$	$\llbracket M N \rrbracket = \llbracket M \rrbracket; \llbracket N \rrbracket; \text{app}$	$\llbracket \text{fst } M \rrbracket = \llbracket M \rrbracket; \text{fst}$
$\llbracket U_i \rrbracket = \text{ty } U_i$	$\llbracket \Sigma x:A. B \rrbracket = \text{ty } \Sigma x:A. B$	$\llbracket \text{snd } M \rrbracket = \llbracket M \rrbracket; \text{snd}$
$\llbracket \Pi x:A. B \rrbracket = \text{ty } \Pi x:A. B$	$\llbracket \langle M, N \rangle \rrbracket = \llbracket M \rrbracket; \llbracket N \rrbracket; \text{pair}$	$\llbracket \text{zero} \rrbracket = \text{lit } 0$
$\llbracket \lambda x:A. M \rrbracket = \text{pushc } \llbracket M \rrbracket_c$	$\llbracket \langle \rangle \rrbracket = \text{unit}$	$\llbracket \text{suc } M \rrbracket = \llbracket M \rrbracket; \text{suc}$
$\llbracket \text{rec}_P(M, (x, p).M', N) \rrbracket = \llbracket N \rrbracket; \text{rec } P \llbracket M \rrbracket_c \llbracket M' \rrbracket_c$		$\llbracket M \rrbracket_c = \llbracket M \rrbracket; \text{ret}$
(a) Term compilation		
$\llbracket \lambda x:A. M \rrbracket_v = \langle \cdot, \llbracket M \rrbracket_c \rangle$	$\llbracket \langle M, N \rangle \rrbracket_v = \langle \llbracket M \rrbracket_v, \llbracket N \rrbracket_v \rangle$	$\llbracket n \rrbracket_v = n \quad \llbracket A \rrbracket_v = A \quad \llbracket \langle \rangle \rrbracket_v = \langle \rangle$
(b) Value compilation		

Fig. 5. Compilation from CC_ω to SECD

LEMMA 2.29 (TYPE PRESERVATION). *If $\Gamma \vdash M : A$, then $\Gamma \vdash \llbracket M \rrbracket_c : \cdot \longrightarrow \cdot :: M$. If $\vdash M : A$ and M is a value, then $\llbracket M \rrbracket_v \models M : A$.*

COROLLARY 2.30 (CORRECTNESS OF WHOLE-PROGRAM COMPILATION). *If $\cdot \vdash M : A$, then $\langle \llbracket M \rrbracket_c, \cdot, \cdot, \cdot \rangle \Downarrow v$ with $v \models M : A$.*

COROLLARY 2.31 (CORRECTNESS OF SEPARATE COMPILATION). *If $\Gamma \vdash M : A$ and have closed CC_ω values $\gamma = M_1, \dots, M_n$ such that $\cdot \vdash \gamma : \Gamma$, then $\langle \llbracket M \rrbracket_c, \llbracket M_1 \rrbracket_v \vdash \dots \vdash \llbracket M_n \rrbracket_v, \cdot, \cdot \rangle \Downarrow v$ with $v \models M[\gamma] : A[\gamma]$.*

2.8 Summary

The dependent SECD machine serves well as a theoretical foundation for dependently typed assembly languages, with type safety, termination, and support for safe separate compilation for type-preserving compilers. However, 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 $\text{ty } A$, and nested instructions like `pushc`. Ideally, these features should be eliminated by first transforming the source language CC_ω to a dependently typed intermediate representation as the specification language, and then to 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].

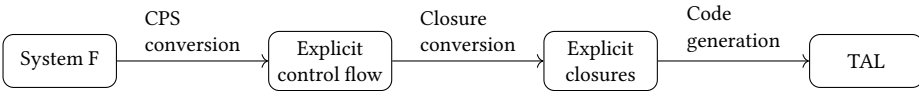
3 Dependent Abstract Machine

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 3.25).

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[]{}.
2   mov r2, 5
3   mov r1, 1
4   jmp fact
5 fact: code[] {r1: int, r2 : int}.
6   bnz r2, nonzero
7   halt[int]

8 nonzero:
9   code[] {r1 : int, r2 : int}.
10  mul r1, r1, r2
11  sub r2, r2, r1
12  jmp fact

```

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 $\text{Single}_A(v)$, whose only inhabitant is v of type A . DTAL [Xi and Harper 2001] used integer-valued singletons $\text{Single}_{\text{int}}(x)$ 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_2 (Lines 2–3) and ℓ_5 (Lines 5–11).

				DAM	
1	mult	$(\{x:\text{Nat}\}, y:\text{Nat} \mapsto x * y : \text{Nat}) :$	12	fact	$(\{\}, y : \text{Nat} \mapsto$
2	ℓ_z	$(\{y:\text{Nat}\}, _:\text{Unit} \mapsto 1 : \text{Nat}) :$			$\text{rec}_{\text{Nat}}(1, (x, p).(\text{suc } x) * p, y) : \text{Nat}) :$
3	lit	1	13	var	y
4	ret		14	rec	P $\ell_z \ell_s$
5	ℓ_s	$(\{y : \text{Nat}, x : \text{Nat}\}, p : \text{Nat} \mapsto$	15	ret	
		$(\text{suc } x) * p : \text{Nat}) :$	16	main	$(\cdot \vdash 120 : \text{Nat}) :$
6	var	x	17	clo	0 fact
7	suc		18	lit	5
8	clo	1 mult	19	app	
9	var	p	20	ret	
10	app				
11	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 $\text{rec}_{\text{Nat}}(1, (x, p).(\text{suc } x) * p, 5)$, 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 3.18).

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

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 $\sigma = M_1, \dots, M_n$ 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
DCC-TY-LABEL	DCC-RED-BETA	
$\frac{\mathfrak{D}; \Gamma \vdash \sigma : \Delta \quad \ell(\{\Delta\}, x:A \mapsto M : B) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \ell\{\sigma\} : \Pi x:A[\sigma]. B[\sigma]}$	$\frac{\ell(\{\Delta\}, x:A \mapsto M : B) \in \mathfrak{D} \quad \mathfrak{D}; \Gamma \vdash \sigma : \Delta \quad \mathfrak{D}; \Gamma \vdash N : A[M/\Delta]}{\mathfrak{D}; \Gamma \vdash \ell\{\sigma\} N \triangleright M[\sigma, N/x] : B[\sigma, N/x]}$	

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
DCC-WF-EMPTY	DCC-WF-LABEL	DCC-WF-TYPE	
$\frac{}{\vdash \cdot; \cdot}$	$\frac{\mathfrak{D}; \Delta, x:A \vdash M : B}{\vdash \mathfrak{D}, \ell(\{\Delta\}, x:A \mapsto N : B); \cdot}$	$\frac{\vdash \mathfrak{D}; \Gamma \quad \mathfrak{D}; \Gamma \vdash A : U_i}{\vdash \mathfrak{D}; \Gamma, x:A}$	

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, $\mathfrak{D}; \Gamma \vdash U_i \equiv \text{Nat} : U$ implies that $U \equiv \text{Nat}$ 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 (SUBJECT REDUCTION). *If $\mathfrak{D}; \Gamma \vdash M \triangleright N : A$, then $\mathfrak{D}; \Gamma \vdash M : A$ and $\mathfrak{D}; \Gamma \vdash N : A$.*

LEMMA 3.2 (WELL-FORMEDNESS OF TYPING JUDGEMENTS). *The parameters of the judgements are well-formed in the following sense:*

- If $\mathcal{D}; \Gamma \vdash M : A$, then $\vdash \mathcal{D}; \Gamma$ and $\mathcal{D}; \Gamma \vdash A : U$.
- If $\mathcal{D}; \Gamma \vdash M \equiv N : A$, then $\mathcal{D}; \Gamma \vdash M : A$ and $\mathcal{D}; \Gamma \vdash N : A$.

LEMMA 3.3 (INJECTIVITY OF TYPE FORMERS). (Postulated) *The type formers are injective in the following sense:*

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

LEMMA 3.4 (NON-CONFUSION OF TYPE FORMERS). (Postulated) *Different type formers cannot be equivalent, specifically:*

- $\mathcal{D}; \Gamma \vdash U_i \equiv \text{Nat} : U$ is not derivable,
- $\mathcal{D}; \Gamma \vdash U_i \equiv \text{Unit} : U$ is not derivable,
- and so on.

THEOREM 3.5 (DEEP NORMALIZATION). *If $\mathcal{D}; \Gamma \vdash M : A$, then M reduces to a β -short normal form.*

THEOREM 3.6 (CONSISTENCY). *There is no $\mathcal{D}$ and M such that $\mathcal{D}; \cdot \vdash M : \Pi A:U_0. A$.*

		DCC	
DCC-TY-APPLY	$\mathcal{D}; \Gamma \vdash M : \Pi x:A. B \quad \mathcal{D}; \Gamma \vdash N : A$	DCC-TY-PAIR	$\mathcal{D}; \Gamma \vdash M : A \quad \mathcal{D}; \Gamma \vdash N : B[M/x]$
	$\mathcal{D}; \Gamma \vdash M N : B[N/x]$		$\mathcal{D}; \Gamma \vdash \langle M, N \rangle : \Sigma x:A. B$
DCC-TY-REC	$\mathcal{D}; \Gamma, x:\text{Nat} \vdash P : U \quad \mathcal{D}; \Gamma \vdash N : \text{Nat}$		
	$\mathcal{D}; \Gamma \vdash M : P[\text{zero}/x] \quad \mathcal{D}; \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{suc } x/x]$		
	$\mathcal{D}; \Gamma \vdash \text{rec}_P(M, (x, p).M', N) : P[N/x]$		
DCC-RED-SND	$\mathcal{D}; \Gamma \vdash M : A \quad \mathcal{D}; \Gamma \vdash N : B[M/x]$	DCC-EQ-ETAPI	$\mathcal{D}; \Gamma \vdash M : \Pi x:A. B \quad \mathcal{D}; \Gamma \vdash N : \Pi x:A. B$
	$\mathcal{D}; \Gamma \vdash \text{snd } \langle M, N \rangle \triangleright N : B[M/x]$		$\mathcal{D}; \Gamma, x:A \vdash M x \equiv N x : B$
			$\mathcal{D}; \Gamma \vdash M \equiv N : \Pi x:A. B$

Fig. 7. Typing judgements, reduction, and equivalence of DCC (excerpt)

3.3 Dependent Abstract Machine

Fig. 8 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 $\text{clo } n \ell$ 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. 9). The operations of DAM have only a

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

Fig. 8. Syntax of the dependent abstract machine (DAM)

						DAM
Rule	Instruction	Env.	Stack	Frame	Condition	
R	ins ins'	env env'	st st'	fr 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	$\text{ty } A ; \text{ins}$ ins	env env	st $\text{st} :: A[\text{env}]$	fr fr		
APP	$\text{app} ; \text{ins}$ ins'	env $\text{env}' :: v$	$\text{st} :: \ell\{\text{env}'\} :: v$ $\cdot$	fr $\text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle$	$\mathfrak{B}(\ell) = \text{ins}'$	
CLO	$\text{clo } n \ell$ ins'	env env	$\text{st} :: \text{st}'$ $\text{st} :: \ell\{\text{st}'\}$	fr fr	$ \text{st}' = n$	
RECZ	$\text{rec } P \ell_z \ell_s ; \text{ins}$ ins'	env $\text{env} :: \langle \rangle$	$\text{st} :: 0$ $\cdot$	fr $\text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle$	$\mathfrak{B}(\ell_z) = \text{ins}'$	

Fig. 9. Operational semantics of DAM (excerpt)

few differences from the SECD machine. $\text{ty } A$ treats the current environment as a substitution and pushes the closed type $A[\text{env}]$ 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.

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 for conciseness.

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  $\ell$` 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  $\ell_z$   $\ell_s$` 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 $\text{rec}_P(M, (x, p).M', N)$ as the result (**DAM-TY-REC**).

DAM-TY-CLO

$$\frac{|\Delta| = n \quad \mathfrak{D}; \Gamma \vdash \sigma' : \Delta \quad \ell(\{\Delta\}, x:A \mapsto M : B) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \text{clo } n \ell : \sigma :: \sigma' \rightarrow \sigma :: \ell\{\sigma'\}}$$

DAM-TY-APP

$$\frac{\mathfrak{D}; \Gamma \vdash M : \Pi x:A. B \quad \mathfrak{D}; \Gamma \vdash N : A}{\mathfrak{D}; \Gamma \vdash \text{app} : \sigma :: M :: N \rightarrow \sigma :: MN}$$

DAM-TY-REC

$$\frac{\begin{array}{l} \mathfrak{D}; \Gamma, x:\text{Nat} \vdash P : U_i \quad \mathfrak{D}; \Gamma \vdash M : P[\text{zero}/x] \quad \mathfrak{D}; \Gamma \vdash N : \text{Nat} \\ \ell_z(\{\Gamma\}, _:\text{Unit} \mapsto M : P[\text{zero}/x]) \in \mathfrak{D} \\ \ell_s(\{\Gamma, x:\text{Nat}\}, p:P \mapsto M' : P[\text{suc } x/x]) \in \mathfrak{D} \end{array}}{\mathfrak{D}; \Gamma \vdash \text{rec } P \ell_z \ell_s : \sigma :: N \rightarrow \sigma :: \text{rec}_P(M, (x, p).M', N)}$$

DAM

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

- (1) $\vdash \mathfrak{B} : \mathfrak{D}$ The code blocks are well-typed with their type signatures $\mathfrak{D}$.
- (2) $\mathfrak{D} \vdash \text{env} \models \Gamma$ The runtime environment has types Γ .
- (3) $\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma$ The runtime stack implements σ , given some env that implements Γ .
- (4) $\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{fr} : [M : A]$ 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	
$\frac{\text{DAM-BLOCKS-NIL}}{\vdash \cdot : \cdot}$	$\frac{\text{DAM-BLOCKS-CONS} \quad \vdash \mathfrak{B} : \mathfrak{D} \quad \mathfrak{D}; \Delta, x:A \vdash M : B \quad \mathfrak{D}; \Delta, x:A \vdash \text{ins} : \cdot \rightarrow \sigma :: M}{\vdash \mathfrak{B}; \ell(\{\Delta\}, x:A \mapsto M : B) : \text{ins} : \mathfrak{D}, \ell(\{\Delta\}, x:A \mapsto M : B)}$

Environment, stack, and frames. The rest of the judgements are again similar to those in dependent SECD. The judgement $\mathfrak{D} \vdash \text{env} \models \Gamma$ 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 $\mathfrak{D} \vdash \text{env} :: v \models \Gamma, x:A$ if $\mathfrak{D} \vdash \text{env} \models \Gamma$ and v has type $A[\text{env}]$.

The judgement for runtime stacks $\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma$ means that st implements the abstract stack σ under runtime environment $\text{env} : \Gamma$. The empty case requires $\mathfrak{D} \vdash \text{env} \models \Gamma$, and $\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} :: v \models \sigma :: M$ if $\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma$ and $\text{val} \equiv M[\text{env}]$, 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	
$\frac{\text{DAM-ENV-CONS} \quad \mathfrak{D} \vdash \text{env} \models \Gamma \quad \mathfrak{D} \vdash v : A[\text{env}]}{\mathfrak{D} \vdash \text{env}, v \models \Gamma, x:A}$	$\frac{\text{DAM-ST-CONS} \quad \mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma \quad \mathfrak{D} \vdash v : A[\text{env}] \quad \mathfrak{D}; \Gamma \vdash M : A \quad \mathfrak{D} \vdash M[\text{env}] \equiv v : A[\text{env}]}{\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st}, v \models \sigma :: M}$

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. Then, we prove progress and preservation by induction on the instruction sequence, and details of the proofs are similar to the proofs for dependent SECD machine. From type safety, we conclude that DAM has partial correctness.

DEFINITION 3.7 (CONFIGURATION WELL-FORMEDNESS). A configuration $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{B}}$ is well-formed, denoted as $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{B}}$, if the following conditions hold:

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

LEMMA 3.8 (WELL-FORMEDNESS OF TYPING JUDGEMENTS). *The parameters of the judgements are well-formed in the following sense:*

- If $\mathcal{D}; \Gamma \vdash \text{ins} : \sigma \rightarrow \sigma'$, then $\vdash \mathcal{D}; \Gamma$ and every term in (σ, σ') is well-typed under $\mathcal{D}; \Gamma$.
- If $\mathcal{D} \vdash \text{env} \models \Gamma$, then $\vdash \mathcal{D}; \Gamma$ and $|\text{env}| = |\Gamma|$.
- If $\mathcal{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma$, then $\mathcal{D} \vdash \text{env} \models \Gamma$, $\mathcal{D}; \Gamma \vdash \sigma : \Delta$, $|\text{st}| = |\sigma|$ and every term in σ is well-typed under $\mathcal{D}; \Gamma$.
- If $\mathcal{D}; (\text{env} : \Gamma) \vdash \text{fr} : [M : A]$, then $\mathcal{D} \vdash \text{env} \models \Gamma$ and $\mathcal{D}; \Gamma \vdash M : A$.

LEMMA 3.9 (CANONICITY OF VALUES). *If $\mathcal{D}; \cdot \vdash v : A$, then*

- If $\mathcal{D} \vdash A \equiv U_i : U_{i+1}$, then $v = A'$ for some type A' .
- If $\mathcal{D} \vdash A \equiv \text{Nat} : U$, then $v = n$ for some natural number n .
- If $\mathcal{D} \vdash A \equiv \text{Unit} : U$, then $v = \langle \rangle$.
- If $\mathcal{D} \vdash A \equiv \Pi x : A'. B' : U$, then $v = \ell\{\text{env}\}$ for some env, ℓ .
- If $\mathcal{D} \vdash A \equiv \Sigma x : A'. B' : U$, then $v = \langle v, v' \rangle$ for some v, v' .

THEOREM 3.10 (PROGRESS). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{B}}$, then either $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{B}} \rightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle_{\mathfrak{B}}$ or $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{B}}$ halts.*

THEOREM 3.11 (PRESERVATION). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{B}}$ and $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{B}} \rightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle_{\mathfrak{B}}$, then $\vdash \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle_{\mathfrak{B}}$.*

COROLLARY 3.12 (TYPE SAFETY). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{B}}$, then it never reduces to a configuration that is irreducible and not at halt.*

COROLLARY 3.13 (PARTIAL CORRECTNESS). *If $\mathcal{D}; \Gamma \vdash \text{ins} : \sigma \rightarrow \sigma' :: M$ with $\mathcal{D}; \Gamma \vdash M : A$, then for all $(\mathfrak{B}, \text{env}, \text{st})$ such that $\vdash \langle \text{ins}, \text{env}, \text{st}, \cdot \rangle_{\mathfrak{B}}$ and $\langle \text{ins}, \text{env}, \text{st}, \cdot \rangle_{\mathfrak{B}} \Downarrow v$, we have $\mathcal{D} \vdash v \equiv M[\text{env}] : A[\text{env}]$.*

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 $\llbracket - \rrbracket_b$ from well-typed DCC terms to CC_{ω} by [Huang and Yallop](#). $\llbracket M \rrbracket_b$ implicitly takes the well-typed derivation of M an argument. It maps $\ell\{\sigma\}$ to $\lambda x : (\llbracket A \rrbracket_b \llbracket \llbracket \sigma \rrbracket_b \rrbracket) . \llbracket N \rrbracket_b \llbracket \llbracket \sigma \rrbracket_b \rrbracket$ given $\ell(\{\Delta\}, x : A \mapsto N : B)$ 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 3.14.

LEMMA 3.14 ($\llbracket - \rrbracket_b$ PRESERVES TYPES). *If $\mathcal{D}; \Gamma \vdash M : A$, then $\llbracket \Gamma \rrbracket_b \vdash \llbracket M \rrbracket_b : \llbracket A \rrbracket_b$.*

$$\begin{aligned}
 \llbracket \text{clo } n \ell \rrbracket_i &= \text{clo } n \llbracket \mathfrak{B}(\ell) \rrbracket_i & \llbracket \text{rec } P \ell_z \ell_s \rrbracket_i &= \text{rec } \llbracket P \rrbracket_b \llbracket \mathfrak{B}(\ell_z) \rrbracket_i \llbracket \mathfrak{B}(\ell_s) \rrbracket_i \\
 \llbracket \ell\{\overline{M}\} \rrbracket_v &= \langle \llbracket \overline{M} \rrbracket_v, \llbracket \mathfrak{B}(\ell) \rrbracket_i \rangle & \llbracket A \rrbracket_v &= \llbracket A \rrbracket_b & \llbracket \langle v, v' \rangle \rrbracket_v &= \langle \llbracket v \rrbracket_v, \llbracket v' \rrbracket_v \rangle \\
 \llbracket \text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle \rrbracket_{fr} &= \llbracket \text{fr} \rrbracket_{fr} :: \langle \llbracket \text{ins} \rrbracket_i, \llbracket \text{env} \rrbracket_v, \llbracket \text{st} \rrbracket_v \rangle \\
 \llbracket \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{B}} \rrbracket_{cf} &= \langle \llbracket \text{ins} \rrbracket_i, \llbracket \text{env} \rrbracket_v, \llbracket \text{st} \rrbracket_v, \llbracket \text{fr} \rrbracket_{fr} \rangle
 \end{aligned}$$

$\llbracket \text{DAM} \rrbracket_{cf}$

Fig. 10. Backward transformation from DAM to dependent SECD (excerpt)

Fig. 10 shows the backward transformation $\llbracket - \rrbracket_i$ 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 $\llbracket - \rrbracket_v$ that transforms DAM values to SECD-machine values. Natural numbers and the unit value remain the same. $\llbracket - \rrbracket_v$ 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 $\llbracket - \rrbracket_v$ pointwise and transform call frames with a combination of $\llbracket - \rrbracket_i$ and $\llbracket - \rrbracket_v$. Finally, we can transform a DAM configuration by applying the respective backward transformations for the four components of the machine. Transformations of the machine $\llbracket - \rrbracket_{cf}$ and its components $\llbracket - \rrbracket_i$, $\llbracket - \rrbracket_v$, and $\llbracket - \rrbracket_{fr}$ 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 3.15). We prove type preservation by proving the following sub-goals by induction:

- If $\mathfrak{D}; \Gamma \vdash \text{ins} : \sigma \rightarrow \sigma'$, then $\llbracket \Gamma \rrbracket_b \vdash \llbracket \text{ins} \rrbracket_i : \llbracket \sigma \rrbracket_b \rightarrow \llbracket \sigma' \rrbracket_b$.
- If $\mathfrak{D} \vdash v : A$, then $\llbracket v \rrbracket_v \models \llbracket v \rrbracket_b : \llbracket A \rrbracket_b$.
- If $\mathfrak{D} \vdash \text{env} \models \Gamma$, then $\llbracket \text{env} \rrbracket_v \models \llbracket \text{env} \rrbracket_b : \llbracket \Gamma \rrbracket_b$.
- If $\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma$, then $\llbracket \text{env} \rrbracket_v : \llbracket \text{env} \rrbracket_b, \llbracket \Gamma \rrbracket_b \vdash \llbracket \text{st} \rrbracket_v \models \llbracket \sigma \rrbracket_b$.
- If $\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{fr} : [M : A]$, then $\llbracket \text{env} \rrbracket_v : \llbracket \text{env} \rrbracket_b, \llbracket \Gamma \rrbracket_b \vdash \llbracket \text{fr} \rrbracket_{fr} : \llbracket [M] \rrbracket_b : \llbracket A \rrbracket_b$.

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. $\llbracket v \rrbracket_v$ implements $\llbracket v \rrbracket_b$.

We also prove that the backward transformation preserves reductions: if a DAM configuration steps to some next state $\langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle_{\mathfrak{D}}$, then the transformed SECD machine configuration steps the transformation of $\langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle_{\mathfrak{D}}$ (Lemma 3.16). Combined with type-preservation, we can prove termination for all well-formed DAM programs by contradiction (Theorem 3.17), and finally conclude that DAM has total correctness (Corollary 3.18) from partial correctness and termination.

LEMMA 3.15 ($\llbracket - \rrbracket_{cf}$ PRESERVES WELL-FORMEDNESS). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{D}}$, then $\vdash \llbracket \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{D}} \rrbracket_{cf}$.*

LEMMA 3.16 ($\llbracket - \rrbracket_{cf}$ PRESERVES REDUCTIONS). *If $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{D}} \rightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle_{\mathfrak{D}}$, then $\llbracket \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{D}} \rrbracket_{cf} \rightarrow \llbracket \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle_{\mathfrak{D}} \rrbracket_{cf}$.*

THEOREM 3.17 (TERMINATION). *If $\vdash \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{D}}$, then $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle_{\mathfrak{D}} \Downarrow v$.*

COROLLARY 3.18 (TOTAL CORRECTNESS). *If $\mathfrak{D}; \Gamma \vdash \text{ins} : \sigma \rightarrow \sigma' :: M$ with $\mathfrak{D}; \Gamma \vdash M : A, \vdash \mathfrak{B} : \mathfrak{D}, \mathfrak{D} \vdash \text{env} \models \Gamma$, and $\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma$, then $\langle \text{ins}, \text{env}, \text{st}, \cdot \rangle_{\mathfrak{D}} \Downarrow v$ with $\mathfrak{D} \vdash v \equiv M[\text{env}] : A[\text{env}]$.*

COROLLARY 3.19 (TOTAL CORRECTNESS FOR PROGRAMS). *If $\mathfrak{D}; \cdot \vdash \text{ins} : \cdot \rightarrow \sigma :: M$ with $\mathfrak{D} \vdash M : A$, then $\langle \text{ins}, \cdot, \cdot, \cdot \rangle_{\mathfrak{D}} \Downarrow v$ 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. 11). Given a DCC derivation, $\text{gen}(\mathfrak{D}; \Gamma \vdash M : A)$ produces an instruction sequence for the expression 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 we produce $\mathfrak{B}$ by generating code for each label in $\mathfrak{D}$.

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

$\llbracket x \rrbracket = \text{var } x$	$\llbracket M \ N \rrbracket = \llbracket M \rrbracket; \llbracket N \rrbracket; \text{app}$	$\llbracket \text{fst } M \rrbracket = \llbracket M \rrbracket; \text{fst}$
$\llbracket U_i \rrbracket = \text{ty } U_i$	$\llbracket \Sigma x:A. B \rrbracket = \text{ty } \Sigma x:A. B$	$\llbracket \text{snd } M \rrbracket = \llbracket M \rrbracket; \text{snd}$
$\llbracket \Pi x:A. B \rrbracket = \text{ty } \Pi x:A. B$	$\llbracket \langle M, N \rangle \rrbracket = \llbracket M \rrbracket; \llbracket N \rrbracket; \text{pair}$	$\llbracket \text{succ } M \rrbracket = \llbracket M \rrbracket; \text{succ}$
$\llbracket \ell \{M_1, \dots, M_n\} \rrbracket = \llbracket M_1 \rrbracket; \dots; \llbracket M_n \rrbracket; \text{clo } n \ \ell$		
$\llbracket \text{rec}_P(M, (x, p).M', N) \rrbracket = \text{rec } P \ \ell_z \ \ell_s$		
$\llbracket \mathcal{D}, \ell(\{\Delta\}, x:A \mapsto M : B) \rrbracket = \llbracket \mathcal{D} \rrbracket; \ell(\{\Delta\}, x:A \mapsto M : B) : \llbracket M \rrbracket$		
$\text{gen}(\mathcal{D}; \Gamma \vdash M : A) = (\llbracket M \rrbracket, \llbracket h_D(\mathcal{D}); h_{\text{exp}}(M) \rrbracket)$		

gen(DCC)

Fig. 11. Code generation for DCC (excerpt)

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 $\mathcal{D}$ and in an expression M , generating an expanded label context $\mathcal{D}'$.

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.

$h_{\text{exp}}(\text{rec}_P(M, (x, p).M', N)) = h_{\text{exp}}(M); h_{\text{exp}}(M'); h_{\text{exp}}(N);$	
$\ell_z(\{\Gamma\}, _ : \text{Unit} \mapsto M : P[\text{zero}/x]);$	
$\ell_s(\{\Gamma, x:\text{Nat}\}, p:P x \mapsto M' : P[\text{succ } x/x]);$	
$h_D(\mathcal{D}, \ell(\{\Delta\}, x:A \mapsto M : B)) = h_D(\mathcal{D}); h_{\text{exp}}(M); \ell(\{\Delta\}, x:A \mapsto M : B)$	

gen(DCC)

Fig. 12. Recursor hoisting (excerpt)

Type preservation. We now show that code generation is type-preserving: The code generation is type-preserving: $\text{gen}(\mathcal{D}; \Gamma \vdash M : A)$ produces an instruction sequence that computes M and a well-formed code block $\mathcal{B}$ that implements $\mathcal{D}$ (Theorem 3.23). It is also correct for whole-program compilation and separate compilation as a corollary of total correctness.

We first define the notion of having recursor labels for a recursor term (Definition 3.20). 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 3.21), and that code generation produces well-typed results if the label context that has labels for all recursors (Lemma 3.22). As a corollary, code generation is type-preserving in the sense of Theorem 3.23, and it is correct for whole-program and separate compilations since DAM has total correctness.

DEFINITION 3.20. $\mathcal{D}$ has recursor labels for a recursor $\mathcal{D}; \Delta \vdash \text{rec}_P(M, (x, p).M', N) : P[N/x]$ if there exist $\ell_z(\{\Delta\}, x:\text{Unit} \mapsto M : P[\text{zero}/x]) \in \mathcal{D}$ and $\ell_s(\{\Delta, x:\text{Nat}\}, p:P \mapsto M' : P[\text{succ } x/x]) \in \mathcal{D}$.

LEMMA 3.21. *Suppose $\mathcal{D}; \Gamma \vdash M : A$, and let $\mathcal{D}' = h_D(\mathcal{D}); h_{\text{exp}}(M)$. Then $\vdash \mathcal{D}', \mathcal{D} \subseteq \mathcal{D}'$, and $\mathcal{D}'$ has recursor labels for all recursor sub-terms in $\mathcal{D}$ and M .*

LEMMA 3.22. *If $\mathcal{D}; \Gamma \vdash M : A$ and $\mathcal{D}$ has recursor labels for all recursor sub-terms in $\mathcal{D}$ and M , then $\mathcal{D}; \Gamma \vdash \llbracket M \rrbracket : \cdot \rightarrow \sigma :: M$ and $\vdash \llbracket \mathcal{D} \rrbracket : \mathcal{D}$.*

PROOF. By induction over the typing derivation of DCC. $\square$

THEOREM 3.23 (TYPE PRESERVATION). *Suppose $\mathcal{D}; \Gamma \vdash M : A$, and let $(\text{ins}, \mathfrak{B}) = \text{gen}(\mathcal{D}; \Gamma \vdash M : A)$. Then $\vdash \mathfrak{B} : \mathcal{D}'$ and $\mathcal{D}'; \Gamma \vdash \text{ins} : \cdot \rightarrow \sigma :: M$.*

COROLLARY 3.24 (CORRECTNESS OF WHOLE-PROGRAM COMPILATION). *Suppose $\mathcal{D}; \cdot \vdash M : A$, and let $(\text{ins}, \mathfrak{B}) = \text{gen}(\mathcal{D}; \cdot \vdash M : A)$. Then, $\langle \text{ins}, \cdot, \cdot, \cdot \rangle_{\mathfrak{B}} \Downarrow v$ with $\mathcal{D} \vdash v \equiv M : A$.*

COROLLARY 3.25 (CORRECTNESS OF SEPARATE COMPILATION). *Suppose $\mathcal{D}; \Gamma \vdash M : A$, and let env be a list of closed DCC values such that $\mathcal{D} \vdash \text{env} : \Gamma$ and $(\text{ins}, \mathfrak{B}) = \text{gen}(\mathcal{D}; \Gamma \vdash M : A)$. Then, $\langle \text{ins}, \text{env}, \cdot, \cdot \rangle_{\mathfrak{B}} \Downarrow v$ with $\mathcal{D} \vdash v \equiv M[\text{env}] : A[\text{env}]$.*

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 DAM as an encoding in Agda and formalized the proof of its type safety, and we are in the process of formalizing the termination and total correctness proofs of dependent SECD and 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]: although they provide a form of dependent types, they do not support large elimination (and it does not appear that they can be straightforwardly extended to do so), and so they are not suitable for our purpose of a target language for the calculus of constructions. 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].

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 that extends System F [Sulzmann et al. 2007], and the whole-program Standard ML compiler MLton makes effective use of typed intermediate representations 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 intermediate languages 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 intermediate language 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 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.

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], 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]. 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. Our type-preserving compilation from DCC to DAM is thus an instance of so-called *translation validation*, in which the correspondence of the semantics of the source and target programs can be straightforwardly mechanically checked after compilation.

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>
- 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 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>
- 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](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](https://doi.org/10.1016/0167-6423(87)90020-7)
- Karl Cray. 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, Louisiana, USA, January 15–17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 198–212. <https://doi.org/10.1145/604131.604149>
- K Cray, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdanczewic. 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>
- 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
- 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>
- 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>
- 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>
- 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 Cray, 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 Cray, 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>
- 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](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>.
- 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), Philadelphia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 181–192. <https://doi.org/10.1145/231379.231414>
- 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>
- 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>

A Complete syntax

A.1 The calculus of constructions (CC)

A.1.1 Syntax.

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

A.1.2 Typing and context well-formedness.

$\Gamma \vdash M : A$

(Typing)

$\frac{\text{TY-UNIVERSE} \quad \vdash \Gamma}{\Gamma \vdash U_i : U_{i+1}}$	$\frac{\text{TY-VAR} \quad x : A \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash x : A}$	$\frac{\text{TY-PI} \quad \Gamma \vdash A : U_i \quad \Gamma, x:A \vdash B : U_j}{\Gamma \vdash \Pi x:A. B : U_{\max(i,j)}}$	$\frac{\text{TY-LAMBDA} \quad \Gamma, x:A \vdash M : B}{\Gamma \vdash \lambda x:A. M : \Pi x:A. B}$
$\frac{\text{TY-APPLY} \quad \Gamma \vdash M : \Pi x:A. B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B[N/x]}$	$\frac{\text{TY-SIGMA} \quad \Gamma \vdash A : U_i \quad \Gamma, x:A \vdash B : U_j}{\Gamma \vdash \Sigma x:A. B : U_{\max(i,j)}}$	$\frac{\text{TY-PAIR} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \langle M, N \rangle : \Sigma x:A. B}$	
$\frac{\text{TY-FST} \quad \Gamma \vdash M : \Sigma x:A. B}{\Gamma \vdash \text{fst } M : A}$	$\frac{\text{TY-SND} \quad \Gamma \vdash M : \Sigma x:A. B}{\Gamma \vdash \text{snd } M : B[\text{fst } M/x]}$	$\frac{\text{TY-NAT} \quad \vdash \Gamma}{\Gamma \vdash \text{Nat} : U_0}$	$\frac{\text{TY-ZERO} \quad \vdash \Gamma}{\Gamma \vdash \text{zero} : \text{Nat}}$
$\frac{\text{TY-SUC} \quad \Gamma \vdash M : \text{Nat}}{\Gamma \vdash \text{suc } M : \text{Nat}}$	$\frac{\text{TY-REC} \quad \Gamma, x:\text{Nat} \vdash P : U \quad \Gamma \vdash N : \text{Nat} \quad \Gamma \vdash M : P[\text{zero}/x] \quad \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{suc } x/x]}{\Gamma \vdash \text{rec}_P(M, (x, p).M', N) : P[N/x]}$	$\frac{\text{TY-UNITTYPE} \quad \vdash \Gamma}{\Gamma \vdash \text{Unit} : U_0}$	
	$\frac{\text{TY-UNIT} \quad \vdash \Gamma}{\Gamma \vdash \langle \rangle : \text{Unit}}$	$\frac{\text{TY-EQUIV} \quad \Gamma \vdash M : A \quad \Gamma \vdash A \equiv B : U}{\Gamma \vdash M : B}$	

$\vdash \Gamma$

(Context well-formedness)

$\frac{\text{WF-EMPTY}}{\vdash \cdot}$	$\frac{\text{WF-CONS} \quad \vdash \Gamma \quad \Gamma \vdash A : U}{\vdash \Gamma, x:A}$
--	---

A.1.3 Reduction and equivalence.

$\Gamma \vdash M \triangleright N : A$

(Reduction)

$\frac{\text{RED-BETA} \quad \Gamma, x:A \vdash N : B \quad \Gamma \vdash M : A}{\Gamma \vdash (\lambda x:A. N) M \triangleright N[M/x] : B[M/x]}$	$\frac{\text{RED-FST} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \text{fst } \langle M, N \rangle \triangleright M : A}$
$\frac{\text{RED-SND} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \text{snd } \langle M, N \rangle \triangleright N : B[M/x]}$	$\frac{\text{RED-RECZ} \quad \Gamma, x:\text{Nat} \vdash P : U \quad \Gamma \vdash M : P[\text{zero}/x] \quad \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{suc } x/x]}{\Gamma \vdash \text{rec}_P(M, (x, p).M', \text{zero}) \triangleright M : P[\text{zero}/x]}$
$\frac{\text{RED-RECS} \quad \Gamma, x:\text{Nat} \vdash P : U \quad \Gamma \vdash N : \text{Nat} \quad \Gamma \vdash M : P[\text{zero}/x] \quad \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{suc } x/x]}{\Gamma \vdash \text{rec}_P(M, (x, p).M', \text{suc } N) \triangleright M' [N/x, \text{rec}_P(M, (x, p).M', N)/p] : P[\text{suc } N/x]}$	

$$\boxed{\Gamma \vdash M \equiv N : A}$$

(Equivalence)

$$\frac{\text{EQ-REDUCE} \quad \Gamma \vdash M \triangleright^* M' : A \quad \Gamma \vdash N \triangleright^* M' : A}{\Gamma \vdash M \equiv N : A} \quad \frac{\text{EQ-ETAPI} \quad \Gamma \vdash M : \Pi x:A. B \quad \Gamma \vdash N : \Pi x:A. B}{\Gamma, x:A \vdash M x \equiv N x : B} \quad \frac{}{\Gamma \vdash M \equiv N : \Pi x:A. B}$$

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

A.1.4 Normal forms of closed types.

$$\boxed{\Vdash A}$$

(Normal forms of closed types)

$$\frac{\text{NF-UNIT} \quad A \equiv_{nf} \text{Unit}}{\Vdash A} \quad \frac{\text{NF-NAT} \quad A \equiv_{nf} \text{Nat}}{\Vdash A} \quad \frac{\text{NF-UNIVERSE} \quad A \equiv_{nf} U}{\Vdash A} \quad \frac{\text{NF-PI} \quad A \equiv_{nf} \Pi x:A'. B' \quad \Vdash A' \quad \forall (a : A'). \Vdash B' [a/x]}{\Vdash A} \quad \frac{\text{NF-SIGMA} \quad A \equiv_{nf} \Sigma x:A'. B' \quad \Vdash A' \quad \forall (a : A'). \Vdash B' [a/x]}{\Vdash A}$$

A.2 Dependent SECD machine

A.2.1 Syntax.

Instructions	l	$::= \text{var } x \mid \text{st } n \mid \text{ty } A \mid \text{pushc ins} \mid \text{app}$ $\mid \text{clo } n \text{ ins} \mid \text{pair} \mid \text{fst} \mid \text{snd}$ $\mid \text{unit} \mid \text{lit } n \mid \text{suc} \mid \text{rec } P \text{ ins ins}'$
Instruction sequences	ins	$::= \text{ret} \mid l; \text{ins}$
Abstract stack	σ	$::= \cdot \mid \sigma :: M$
Values	v	$::= n \mid \langle \rangle \mid A \mid \langle v, v' \rangle \mid \langle \text{env}, \text{ins} \rangle$
Runtime environment/stack	env, st	$::= \cdot \mid \text{env} :: v$
Call frames	fr	$::= \cdot \mid \text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle$
Machine configuration	C	$::= \langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle$

A.2.2 Operational semantics. See Fig. 13.

A.2.3 Instruction typing.

Rule	Instruction	Env.	Stack	Frame
R	ins ins'	env env'	st st'	fr fr'
means that $\langle \text{ins}, \text{env}, \text{st}, \text{fr} \rangle \longrightarrow \langle \text{ins}', \text{env}', \text{st}', \text{fr}' \rangle$ for rule SECD-RED-R.				
VAR	var x ; ins ins	env env	st st :: env(x)	fr fr
ST	st n ; ins ins	env env	st st :: st(n)	fr fr
TY	ty A ; ins ins	env env	st st :: $A[\gamma]$	fr fr
given a proof that env implements a closing substitution γ .				
PUSHC	pushc ins ; ins' ins'	env env	st st :: $\langle \text{env}, \text{ins} \rangle$	fr fr
APP	app ; ins ins'	env env' :: v	st :: $\langle \text{env}', \text{ins}' \rangle$:: v ·	fr fr :: $\langle \text{ins}, \text{env}, \text{st} \rangle$
CLO	clo n ins ; ins' ins'	env env	st :: $v_1, \dots, v_n$ st :: $\langle v_1 :: \dots :: v_n, \text{ins} \rangle$	fr fr
PAIR	pair ; ins ins	env env	st :: v :: v' st :: $\langle v, v' \rangle$	fr fr
FST	fst ; ins ins	env env	st :: $\langle v, v' \rangle$ st :: v	fr fr
SND	snd ; ins ins	env env	st :: $\langle v, v' \rangle$ st :: v'	fr fr
UNIT	unit ; ins ins	env env	st st :: $\langle \rangle$	fr fr
LIT	lit n ; ins ins	env env	st st :: n	fr fr
SUC	suc ; ins ins	env env	st :: n st :: $(n + 1)$	fr fr
RECZ	rec P ins _z ins _s ; ins ins _z	env env	st :: 0 ·	fr fr :: $\langle \text{ins}, \text{env}, \text{st} \rangle$
RECS	rec P ins _z ins _s ; ins rec P ins _z ins _s ; app ; ret	env env	st :: $(n + 1)$ $\langle \text{env} :: n, \text{ins}_s \rangle$:: n	fr fr :: $\langle \text{ins}, \text{env}, \text{st} \rangle$
RET	ret ins'	env env'	st :: v st' :: v	fr :: $\langle \text{ins}', \text{env}', \text{st}' \rangle$ fr

Fig. 13. Operational semantics of dependent SECD

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

(Instruction typing)

$$\frac{\text{SECD-TY-VAR} \quad \vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash \text{var } x : \sigma \rightarrow \sigma :: x}$$

$$\frac{\text{SECD-TY-ST} \quad \vdash \Gamma \quad n < |\sigma| \quad \sigma(n) = M}{\Gamma \vdash \text{st } n : \sigma \rightarrow \sigma :: M}$$

$$\frac{\text{SECD-TY-TY} \quad \Gamma \vdash A : U}{\Gamma \vdash \text{ty } A : \sigma \rightarrow \sigma :: A}$$

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

$$\frac{\text{SECD-TY-APP} \quad \Gamma \vdash M : \Pi x:A. B \quad \Gamma \vdash N : A}{\Gamma \vdash \text{app} : \sigma :: M \rightarrow \sigma :: M N}$$

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

$$\frac{\text{SECD-TY-PAIR} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \text{pair} : \sigma :: M :: N \rightarrow \sigma :: \langle M, N \rangle}$$

$$\frac{\text{SECD-TY-FST} \quad \Gamma \vdash M : \Sigma x:A. B}{\Gamma \vdash \text{fst} : \sigma :: M \rightarrow \sigma :: \text{fst } M}$$

$$\frac{\text{SECD-TY-SND} \quad \Gamma \vdash M : \Sigma x:A. B}{\Gamma \vdash \text{snd} : \sigma :: M \rightarrow \sigma :: \text{snd } M}$$

$$\frac{\text{SECD-TY-UNIT} \quad \vdash \Gamma}{\Gamma \vdash \text{unit} : \sigma \rightarrow \sigma :: \langle \rangle}$$

$$\frac{\text{SECD-TY-LIT} \quad \vdash \Gamma}{\Gamma \vdash \text{lit } n : \sigma \rightarrow \sigma :: n}$$

$$\frac{\text{SECD-TY-SUC} \quad \Gamma \vdash M : \text{Nat}}{\Gamma \vdash \text{suc} : \sigma :: M \rightarrow \sigma :: \text{suc } M}$$

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

$$\frac{\text{SECD-TY-EQUIV} \quad \begin{array}{c} \Gamma \vdash \sigma_1 \equiv \sigma'_1 \quad \Gamma \vdash \sigma_2 \equiv \sigma'_2 \\ \Gamma \vdash \text{ins} : \sigma_1 \rightarrow \sigma_2 \end{array}}{\Gamma \vdash \text{ins} : \sigma'_1 \rightarrow \sigma'_2}$$

$$\boxed{\Gamma \vdash \text{ins} : \sigma \rightarrow \sigma'}$$

(Instruction sequence typing)

$$\frac{\text{SECD-TY-RET} \quad \Gamma \vdash M : A}{\Gamma \vdash \text{ret} : \sigma :: M \rightarrow \sigma :: M}$$

$$\frac{\text{SECD-TY-SEQ} \quad \begin{array}{c} \Gamma \vdash l : \sigma \rightarrow \sigma' \\ \Gamma \vdash \text{ins} : \sigma' \rightarrow \sigma'' \end{array}}{\Gamma \vdash l ; \text{ins} : \sigma \rightarrow \sigma''}$$

A.2.4 Machine well-formedness.

$$\boxed{\text{env} \models \gamma : \Gamma}$$

(Environment typing)

$$\frac{\text{SECD-ENV-NIL}}{\cdot \models \epsilon : \cdot}$$

$$\frac{\text{SECD-ENV-CONS} \quad \begin{array}{c} \Gamma \vdash A : U \\ \text{env} \models \gamma : \Gamma \quad v \models M : A[\gamma] \end{array}}{\text{env} :: v \models \gamma, M/x : \Gamma, x:A}$$

$$\boxed{v \models M : A}$$

(Value typing)

$$\frac{\text{SECD-VAL-LIT}}{n \models n : \text{Nat}}$$

$$\frac{\text{SECD-VAL-UNIT}}{\langle \rangle \models \langle \rangle : \text{Unit}}$$

$$\frac{\text{SECD-VAL-TY} \quad \vdash A : U}{A \models A : U}$$

$$\frac{\text{SECD-VAL-CLO} \quad \begin{array}{c} \Gamma, x:A \vdash M : B \quad \text{env} \models \gamma : \Gamma \\ \Gamma, x:A \vdash \text{ins} : \cdot \rightarrow \sigma :: M \end{array}}{\langle \text{env}, \text{ins} \rangle \models (\lambda x:A. M)[\gamma] : (\Pi x:A. B)[\gamma]}$$

$$\frac{\text{SECD-VAL-PAIR} \quad \begin{array}{c} x:A \vdash B : U \\ v \models M : A \quad v' \models N : B[M/x] \end{array}}{\langle v, v' \rangle \models \langle M, N \rangle : \Sigma x:A. B}$$

$$\boxed{(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma}$$

(Stack typing)

$$\frac{\text{SECD-ST-NIL} \quad \text{env} \models \gamma : \Gamma}{(\text{env} ;_{\gamma} \Gamma) \vdash \cdot \models \cdot} \quad \frac{\text{SECD-ST-CONS} \quad \begin{array}{l} (\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma \quad v \models M' : A[\gamma] \\ \Gamma \vdash M : A \quad \vdash M[\gamma] \equiv M' : A[\gamma] \end{array}}{(\text{env} ;_{\gamma} \Gamma) \vdash \text{st} :: v \models \sigma :: M}$$

$$\boxed{(\text{env} ;_{\gamma} \Gamma) \vdash \text{fr} : [M : A]}$$

(Frame typing)

$$\frac{\text{SECD-FR-NIL} \quad \vdash M : A}{(\cdot ;_{\epsilon} \cdot) \vdash \cdot : [M : A]} \quad \frac{\text{SECD-FR-CONS} \quad \begin{array}{l} \Gamma \vdash \text{ins} : \sigma :: M \rightarrow \sigma' :: N \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B \\ \text{env} \models \gamma : \Gamma \quad (\text{env} ;_{\gamma} \Gamma) \vdash \text{st} \models \sigma \quad (\text{env}' ;_{\gamma'} \Gamma') \vdash \text{fr} : [N' : B'] \\ \vdash B[\gamma] \equiv B'[\gamma'] : U \quad \vdash N[\gamma] \equiv N'[\gamma'] : B[\gamma] \end{array}}{(\text{env} ;_{\gamma} \Gamma) \vdash \text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle : [M : A]}$$

A.3 Defunctionalized CC

A.3.1 Syntax.

$$\begin{array}{ll} \text{Expressions} & A, B, M, N, P ::= x \mid U_i \mid \Pi x:A. B \mid M N \mid \ell\{\overline{M}\} \\ & \quad \mid \Sigma x:A. B \mid \langle M, N \rangle \mid \text{fst } M \mid \text{snd } M \\ & \quad \mid \text{Unit} \mid \langle \rangle \mid \text{Nat} \mid \text{zero} \mid \text{suc } M \mid \text{rec}_P(M, (x, p).M', N) \\ \text{Type contexts} & \Gamma ::= \cdot \mid \Gamma, x:A \\ \text{Label contexts} & \mathcal{D} ::= \cdot \mid \mathcal{D}, \ell(\{\Gamma\}, x:A \mapsto M : B) \end{array}$$

A.3.2 Typing and context well-formedness.

$$\boxed{\mathcal{D}; \Gamma \vdash M : A}$$

(Typing)

$$\begin{array}{c} \frac{\text{DCC-TY-UNIVERSE} \quad \vdash \mathcal{D}; \Gamma}{\mathcal{D}; \Gamma \vdash U_i : U_{i+1}} \quad \frac{\text{DCC-TY-VAR} \quad \mathcal{D} \vdash x : A \in \Gamma \quad \vdash \mathcal{D}; \Gamma}{\mathcal{D}; \Gamma \vdash x : A} \quad \frac{\text{DCC-TY-PI} \quad \mathcal{D}; \Gamma \vdash A : U_i \quad \mathcal{D}; \Gamma, x:A \vdash B : U_j}{\mathcal{D}; \Gamma \vdash \Pi x:A. B : U_{\max(i,j)}} \\ \\ \frac{\text{DCC-TY-LABEL} \quad \begin{array}{l} \mathcal{D}; \Gamma \vdash \sigma : \Delta \\ \ell(\{\Delta\}, x:A \mapsto M : B) \in \mathcal{D} \end{array}}{\mathcal{D}; \Gamma \vdash \ell\{\sigma\} : \Pi x:A[\sigma]. B[\sigma]} \quad \frac{\text{DCC-TY-APPLY} \quad \mathcal{D}; \Gamma \vdash M : \Pi x:A. B \quad \mathcal{D}; \Gamma \vdash N : A}{\mathcal{D}; \Gamma \vdash M N : B[N/x]} \\ \\ \frac{\text{DCC-TY-SIGMA} \quad \mathcal{D}; \Gamma \vdash A : U_i \quad \mathcal{D}; \Gamma, x:A \vdash B : U_j}{\mathcal{D}; \Gamma \vdash \Sigma x:A. B : U_{\max(i,j)}} \quad \frac{\text{DCC-TY-PAIR} \quad \mathcal{D}; \Gamma \vdash M : A \quad \mathcal{D}; \Gamma \vdash N : B[M/x]}{\mathcal{D}; \Gamma \vdash \langle M, N \rangle : \Sigma x:A. B} \\ \\ \frac{\text{DCC-TY-FST} \quad \mathcal{D}; \Gamma \vdash M : \Sigma x:A. B}{\mathcal{D}; \Gamma \vdash \text{fst } M : A} \quad \frac{\text{DCC-TY-SND} \quad \mathcal{D}; \Gamma \vdash M : \Sigma x:A. B}{\mathcal{D}; \Gamma \vdash \text{snd } M : B[\text{fst } M/x]} \quad \frac{\text{DCC-TY-NAT} \quad \vdash \mathcal{D}; \Gamma}{\mathcal{D}; \Gamma \vdash \text{Nat} : U_0} \quad \frac{\text{DCC-TY-ZERO} \quad \vdash \mathcal{D}; \Gamma}{\mathcal{D}; \Gamma \vdash \text{zero} : \text{Nat}} \end{array}$$

$$\begin{array}{c}
\text{DCC-TY-SUC} \\
\frac{\mathfrak{D}; \Gamma \vdash M : \text{Nat}}{\mathfrak{D}; \Gamma \vdash \text{succ } M : \text{Nat}}
\end{array}
\quad
\begin{array}{c}
\text{DCC-TY-REC} \\
\frac{\mathfrak{D}; \Gamma, x:\text{Nat} \vdash P : \mathbb{U} \quad \mathfrak{D}; \Gamma \vdash N : \text{Nat} \quad \mathfrak{D}; \Gamma \vdash M : P[\text{zero}/x] \quad \mathfrak{D}; \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{succ } x/x]}{\mathfrak{D}; \Gamma \vdash \text{rec}_P(M, (x, p).M', N) : P[N/x]}
\end{array}$$

$$\begin{array}{c}
\text{DCC-TY-UNITTYPE} \\
\frac{\vdash \mathfrak{D}; \Gamma}{\mathfrak{D}; \Gamma \vdash \text{Unit} : \mathbb{U}_0}
\end{array}$$

$$\begin{array}{c}
\text{DCC-TY-UNIT} \\
\frac{\vdash \mathfrak{D}; \Gamma}{\mathfrak{D}; \Gamma \vdash \langle \rangle : \text{Unit}}
\end{array}$$

$$\begin{array}{c}
\text{DCC-TY-EQUIV} \\
\frac{\mathfrak{D}; \Gamma \vdash M : A \quad \mathfrak{D}; \Gamma \vdash B : \mathbb{U} \quad \mathfrak{D}; \Gamma \vdash A \equiv B : \mathbb{U}}{\mathfrak{D}; \Gamma \vdash M : B}
\end{array}$$

$$\boxed{\vdash \mathfrak{D}; \Gamma}$$

(Context well-formedness)

$$\begin{array}{c}
\text{DCC-WF-EMPTY} \\
\frac{}{\vdash \cdot;}
\end{array}$$

$$\begin{array}{c}
\text{DCC-WF-LABEL} \\
\frac{\mathfrak{D}; \Delta, x:A \vdash M : B}{\vdash \mathfrak{D}, \ell(\{\Delta\}, x:A \mapsto N : B); \cdot}
\end{array}$$

$$\begin{array}{c}
\text{DCC-WF-TYPE} \\
\frac{\vdash \mathfrak{D}; \Gamma \quad \mathfrak{D}; \Gamma \vdash A : \mathbb{U}_i}{\vdash \mathfrak{D}; \Gamma, x:A}
\end{array}$$

A.3.3 Reduction and equivalence.

$$\boxed{\mathfrak{D}; \Gamma \vdash M \triangleright N : A}$$

(Reduction)

$$\begin{array}{c}
\text{DCC-RED-BETA} \\
\frac{\ell(\{\Delta\}, x:A \mapsto M : B) \in \mathfrak{D} \quad \mathfrak{D}; \Gamma \vdash \sigma : \Delta \quad \mathfrak{D}; \Gamma \vdash N : A[M/\Delta]}{\mathfrak{D}; \Gamma \vdash \ell\{\sigma\} N \triangleright M[\sigma, N/x] : B[\sigma, N/x]}
\end{array}$$

$$\begin{array}{c}
\text{DCC-RED-FST} \\
\frac{\mathfrak{D}; \Gamma \vdash M : A \quad \mathfrak{D}; \Gamma \vdash N : B[M/x]}{\mathfrak{D}; \Gamma \vdash \text{fst } \langle M, N \rangle \triangleright M : A}
\end{array}$$

$$\begin{array}{c}
\text{DCC-RED-SND} \\
\frac{\mathfrak{D}; \Gamma \vdash M : A \quad \mathfrak{D}; \Gamma \vdash N : B[M/x]}{\mathfrak{D}; \Gamma \vdash \text{snd } \langle M, N \rangle \triangleright N : B[M/x]}
\end{array}$$

$$\begin{array}{c}
\text{DCC-RED-RECZ} \\
\frac{\mathfrak{D}; \Gamma, x:\text{Nat} \vdash P : \mathbb{U} \quad \mathfrak{D}; \Gamma \vdash M : P[\text{zero}/x] \quad \mathfrak{D}; \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{succ } x/x]}{\mathfrak{D}; \Gamma \vdash \text{rec}_P(M, (x, p).M', \text{zero}) \triangleright M : P[\text{zero}/x]}
\end{array}$$

$$\begin{array}{c}
\text{DCC-RED-RECS} \\
\frac{\mathfrak{D}; \Gamma, x:\text{Nat} \vdash P : \mathbb{U} \quad \mathfrak{D}; \Gamma \vdash N : \text{Nat} \quad \mathfrak{D}; \Gamma \vdash M : P[\text{zero}/x] \quad \mathfrak{D}; \Gamma, x:\text{Nat}, p:P \vdash M' : P[\text{succ } x/x]}{\mathfrak{D}; \Gamma \vdash \text{rec}_P(M, (x, p).M', \text{succ } N) \triangleright M'[N/x, \text{rec}_P(M, (x, p).M', N)/p] : P[\text{succ } N/x]}
\end{array}$$

$$\boxed{\mathfrak{D}; \Gamma \vdash M \equiv N : A}$$

(Equivalence)

$$\begin{array}{c}
\text{DCC-EQ-REDUCE} \\
\frac{\mathfrak{D}; \Gamma \vdash M \triangleright^* M' : A \quad \mathfrak{D}; \Gamma \vdash N \triangleright^* M' : A}{\mathfrak{D}; \Gamma \vdash M \equiv N : A}
\end{array}$$

$$\begin{array}{c}
\text{DCC-EQ-ETAPI} \\
\frac{\mathfrak{D}; \Gamma \vdash M : \prod x:A. B \quad \mathfrak{D}; \Gamma \vdash N : \prod x:A. B \quad \mathfrak{D}; \Gamma, x:A \vdash M x \equiv N x : B}{\mathfrak{D}; \Gamma \vdash M \equiv N : \prod x:A. B}
\end{array}$$

$$\begin{array}{c}
\text{DCC-EQ-ETASIGMA} \\
\frac{\mathfrak{D}; \Gamma \vdash M : \sum x:A. B}{\mathfrak{D}; \Gamma \vdash \langle \text{fst } M, \text{snd } M \rangle \equiv M : \sum x:A. B}
\end{array}$$

A.4 Dependent abstract machine (DAM)

A.4.1 Syntax.

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

A.4.2 *Operational semantics.* See Fig. 14.

A.4.3 Instruction typing.

$\boxed{\mathfrak{D}; \Gamma \vdash I : \sigma \rightarrow \sigma'}$			<i>(Instruction typing)</i>
DAM-TY-VAR $\frac{\mathfrak{D} \vdash x : A \in \Gamma \quad \vdash \mathfrak{D}; \Gamma}{\mathfrak{D}; \Gamma \vdash \text{var } x : \sigma \rightarrow \sigma :: x}$	DAM-TY-ST $\frac{\vdash \mathfrak{D}; \Gamma \quad \sigma(n) = M \quad n < \sigma }{\mathfrak{D}; \Gamma \vdash \text{st } n : \sigma \rightarrow \sigma :: M}$	DAM-TY-TY $\frac{\mathfrak{D}; \Gamma \vdash A : U}{\mathfrak{D}; \Gamma \vdash \text{ty } A : \sigma \rightarrow \sigma :: A}$	
DAM-TY-APP $\frac{\mathfrak{D}; \Gamma \vdash M : \Pi x:A. B \quad \mathfrak{D}; \Gamma \vdash N : A}{\mathfrak{D}; \Gamma \vdash \text{app} : \sigma :: M :: N \rightarrow \sigma :: M N}$	DAM-TY-CLO $\frac{ \Delta = n \quad \mathfrak{D}; \Gamma \vdash \sigma' : \Delta \quad \ell(\{\Delta\}, x:A \mapsto M : B) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \text{clo } n \ell : \sigma :: \sigma' \rightarrow \sigma :: \ell\{\sigma'\}}$		
DAM-TY-PAIR $\frac{\mathfrak{D}; \Gamma, x:A \vdash B : U_i \quad \mathfrak{D}; \Gamma \vdash M : A \quad \mathfrak{D}; \Gamma \vdash N : B[M/x]}{\mathfrak{D}; \Gamma \vdash \text{pair} : \sigma :: M :: N \rightarrow \sigma :: \langle M, N \rangle}$	DAM-TY-FST $\frac{\mathfrak{D}; \Gamma \vdash M : \Sigma x:A. B}{\mathfrak{D}; \Gamma \vdash \text{fst} : \sigma :: M \rightarrow \sigma :: \text{fst } M}$		
DAM-TY-SND $\frac{\mathfrak{D}; \Gamma \vdash M : \Sigma x:A. B}{\mathfrak{D}; \Gamma \vdash \text{snd} : \sigma :: M \rightarrow \sigma :: \text{snd } M}$	DAM-TY-UNIT $\frac{\vdash \mathfrak{D}; \Gamma}{\mathfrak{D}; \Gamma \vdash \text{unit} : \sigma \rightarrow \sigma :: \langle \rangle}$	DAM-TY-LIT $\frac{\vdash \mathfrak{D}; \Gamma}{\mathfrak{D}; \Gamma \vdash \text{lit } n : \sigma \rightarrow \sigma :: n}$	
	DAM-TY-SUC $\frac{\mathfrak{D}; \Gamma \vdash N : \text{Nat}}{\mathfrak{D}; \Gamma \vdash \text{suc} : \sigma :: N \rightarrow \sigma :: \text{suc } N}$		
DAM-TY-REC $\frac{\mathfrak{D}; \Gamma, x:\text{Nat} \vdash P : U_i \quad \mathfrak{D}; \Gamma \vdash M : P[\text{zero}/x] \quad \mathfrak{D}; \Gamma \vdash N : \text{Nat} \quad \ell_z(\{\Gamma\}, _:\text{Unit} \mapsto M : P[\text{zero}/x]) \in \mathfrak{D} \quad \ell_s(\{\Gamma, x:\text{Nat}\}, p:P \mapsto M' : P[\text{suc } x/x]) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \text{rec } P \ell_z \ell_s : \sigma :: N \rightarrow \sigma :: \text{rec}_P(M, (x, p).M', N)}$	DAM-TY-EQUIV $\frac{\mathfrak{D}; \Gamma \vdash \sigma_1 \equiv \sigma'_1 \quad \mathfrak{D}; \Gamma \vdash \sigma_2 \equiv \sigma'_2}{\mathfrak{D}; \Gamma \vdash \text{ins} : \sigma_1 \rightarrow \sigma_2 \quad \mathfrak{D}; \Gamma \vdash \text{ins} : \sigma'_1 \rightarrow \sigma'_2}$		

Rule	Instruction	Env.	Stack	Frame	Condition
R	ins ins'	env env'	st st'	fr 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.					
VAR	var x ; ins ins	env env	st st :: v	fr fr	$\text{env}(x) = v$
ST	st n ; ins ins	env env	st st :: v	fr fr	$\text{st}(n) = v$
TY	ty A ; ins ins	env env	st st :: $A[\text{env}]$	fr fr	
APP	app; ins ins'	env env' :: v	st :: $\ell\{\text{env}'\} :: v$ ·	fr fr :: $\langle \text{ins}, \text{env}, \text{st} \rangle$	$\mathfrak{B}(\ell) = \text{ins}'$
CLO	clo $n \ell$; ins' ins'	env env	st :: st' st :: $\ell\{\text{st}'\}$	fr fr	$ \text{st}' = n$
PAIR	pair; ins ins	env env	st :: v :: v' st :: $\langle v, v' \rangle$	fr fr	
FST	fst; ins ins	env env	st :: $\langle v, v' \rangle$ st :: v	fr fr	
SND	snd; ins ins	env env	st :: $\langle v, v' \rangle$ st :: v'	fr fr	
UNIT	unit; ins ins	env env	st st :: $\langle \rangle$	fr fr	
LIT	lit n ; ins ins	env env	st st :: n	fr fr	
SUC	suc; ins ins	env env	st :: n st :: $(n + 1)$	fr fr	
RECZ	rec $P \ell_z \ell_s$; ins ins'	env env :: $\langle \rangle$	st :: 0 ·	fr fr :: $\langle \text{ins}, \text{env}, \text{st} \rangle$	$\mathfrak{B}(\ell_z) = \text{ins}'$
RECS	rec $P \ell_z \ell_s$; ins rec $P \ell_z \ell_s$; app; ret	env env	st :: $(n + 1)$ $\ell_s\{\text{env} :: n\} :: n$	fr fr :: $\langle \text{ins}, \text{env}, \text{st} \rangle$	
RET	ret ins'	env env'	st :: v st' :: v	fr :: $\langle \text{ins}', \text{env}', \text{st}' \rangle$ fr	

Fig. 14. Operational semantics of dependent abstract machine

$\mathfrak{D}; \Gamma \vdash \text{ins} : \sigma \rightarrow \sigma'$	(Instruction sequence typing)
DAM-TY-RET	DAM-TY-SEQ
$\mathfrak{D}; \Gamma \vdash \text{ret} : \sigma :: M \rightarrow \sigma :: M$	$\mathfrak{D}; \Gamma \vdash \text{ins} : \sigma \rightarrow \sigma' \quad \mathfrak{D}; \Gamma \vdash \text{ins}' : \sigma' \rightarrow \sigma''$
	$\mathfrak{D}; \Gamma \vdash \text{ins}; \text{ins}' : \sigma \rightarrow \sigma''$

A.4.4 Machine well-formedness.

$\mathfrak{D} \vdash \text{env} \models \Gamma$	(Environment typing)
DAM-ENV-NIL	DAM-ENV-CONS
$\mathfrak{D} \vdash \cdot \models \cdot$	$\mathfrak{D} \vdash \text{env} \models \Gamma \quad \mathfrak{D} \vdash v : A[\text{env}]$
	$\mathfrak{D} \vdash \text{env}, v \models \Gamma, x:A$

$$\boxed{\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma}$$

(Stack typing)

DAM-ST-NIL

$$\frac{}{\mathfrak{D}; (\text{env} : \Gamma) \vdash \cdot \models \cdot}$$

DAM-ST-CONS

$$\frac{\begin{array}{l} \mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma \quad \mathfrak{D} \vdash v : A[\text{env}] \\ \mathfrak{D}; \Gamma \vdash M : A \quad \mathfrak{D} \vdash M[\text{env}] \equiv v : A[\text{env}] \end{array}}{\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st}, v \models \sigma :: M}$$

$$\boxed{\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{fr} : [M : A]}$$

(Frame typing)

DAM-FR-CONS

$$\frac{\begin{array}{l} \mathfrak{D}; (\text{env}' : \Gamma') \vdash \text{fr} : [N' : B'] \\ \mathfrak{D}; \Gamma \vdash \text{ins} : \sigma :: M \rightarrow \sigma' :: N \\ \mathfrak{D} \vdash \text{env} \models \Gamma \quad \mathfrak{D}; (\text{env} : \Gamma) \vdash \text{st} \models \sigma \\ \mathfrak{D}; \Gamma \vdash M : A \quad \mathfrak{D}; \Gamma \vdash N : B \\ \mathfrak{D} \vdash B[\text{env}] \equiv B'[\text{env}] : U \\ \mathfrak{D} \vdash N[\text{env}] \equiv N'[\text{env}'] : B[\text{env}] \end{array}}{\mathfrak{D}; (\text{env} : \Gamma) \vdash \text{fr} :: \langle \text{ins}, \text{env}, \text{st} \rangle : [M : A]}$$

DAM-FR-NIL

$$\frac{\mathfrak{D} \vdash v : A}{\mathfrak{D}; (\cdot : \cdot) \vdash \cdot : [v : A]}$$

$$\boxed{\vdash \mathfrak{B} : \mathfrak{D}}$$

(Code block typing)

DAM-BLOCKS-NIL

$$\frac{}{\vdash \cdot : \cdot}$$

DAM-BLOCKS-CONS

$$\frac{\begin{array}{l} \vdash \mathfrak{B} : \mathfrak{D} \quad \mathfrak{D}; \Delta, x:A \vdash M : B \quad \mathfrak{D}; \Delta, x:A \vdash \text{ins} : \cdot \rightarrow \sigma :: M \end{array}}{\vdash \mathfrak{B}; \ell(\{\Delta\}, x:A \mapsto M : B) : \text{ins} : \mathfrak{D}, \ell(\{\Delta\}, x:A \mapsto M : B)}$$