QTAL: A quantitatively and dependently typed assembly language

First-year report for CPGS in Computer Science

Department of Computer Science and Technology
University of Cambridge
Oct 2023

Yulong Huang

Supervised by Dr Jeremy Yallop
and Prof. Marcelo Fiore

1 Introduction

Dependent types are the most sophisticated types available in programming languages. They allow programmers to specify a function's properties precisely with logical statements, and the type system ensures that such properties hold for well-typed programs. For example, we can state that a function contains no out-of-bound array access, an implementation of merge sort correctly sorts a list, and server code fully respect complex network protocols. The reliability and security of dependently typed programs are becoming increasingly important to our complicated large-scale projects where correctness is critical. Alas, the performance of existing compilers' generated output is no match to our practical need.

To see this more precisely, take a look at the following comparison between the usual and the dependently typed implementations of lists, polymorphic over some type `A`. The dependent version at the right is often referred to as vectors.

```
data List A : Type
  [] : List A
  Cons : A → List A → List A
```

```
data Vec A : (n : Nat) → Type
  [] : List A 0
  Cons : A → Vec A n → Vec A (suc n)
```

As you can see, two implementations have the same constructors, but `Vec` has a *type index*, a natural number `n` that tracks its length. Empty lists have zero length and `Cons` adds the length by one. Function types may depend on the type indices (as in `repN`) and perform computations on them (as in `append`).

```
(* repN x n creates a list of n copies of x *)
repN : ∀{A} → (x : A) → (n : Nat) → Vec A n
(* compiler infers implicit arguments {A m n} automatically *)
append : ∀{A m n} → Vec A m → Vec A n → Vec A (m + n)
```

Using the *propositions-as-types* analogy, we view dependent types as predicates over their indices. We can define an indexed type `LessThan x y` over two natural numbers and its inhabitants represent proofs of `x` being less than `y`. Functions can take an argument of this type as a constraint, for example, the `lookup` below that finds an element at position `i` from a vector of length `n`. By requiring a proof of `LessThan i n`, `lookup` promises no index-out-of-bound errors.

```
lookup : ∀{n} → Vec A n → (i : Nat) → LessThan i n → A
```

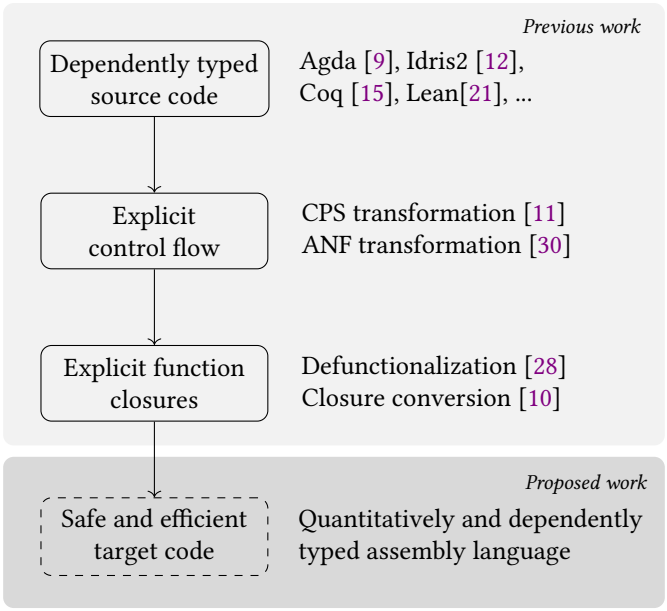


Figure 1: Current development of type-preserving compilers for dependent types

What if we want to compile and run dependently typed programs? The performance would be dreadful, if we naively turn it into untyped assembly. Computationally, the vector implementation above does the same operations as the non-dependent lists, but it pays more space to store the type indices and wastes cycles to compute proof terms. It carries the extra work introduced for correctness checking at compile time to runtime, yet throws away the type system's safety guarantee by compiling into untyped target code, making the output fragile to linking errors and compiler-induced bugs. In other words, we get inefficient and possibly unsafe code!

An ideal compiler is type-preserving and supports erasure. It should transform source code into a dependently-typed assembly language to enable correctness checking for the generated target code, and remove the computationally irrelevant parts before execution to obtain performance comparable or even better than non-dependent implementations (since we can safely eliminate runtime bound checks).

We are only one step away from such ideal compilers. As shown in Fig. 1, previous work established well-studied type-preserving code transformations that successively remove abstraction to turn high-level source code into low-level intermediate representations. Erasure is achieved in current dependently typed languages with Quantitative type theory (QTT) [4], which allows users to mark code with usage information that indicates runtime-irrelevance and helps compilers to perform reliable memory-usage optimizations. The last piece required now is a typed assembly language that incorporates both dependent and quantitative types, and is possible to generate from the resulting intermediate code of type-preserving transformations.

My proposed project fills in this last gap with a quantitatively and dependently typed assembly language, which will serve as the target language of future compilers, making it possible for the first time to build high-assurance compilers for languages with extremely sophisticated type systems.

The remaining of this document is structured as follows. I first cover the prerequisite knowledge in dependent type theory and its compilation in §2. Then, I give a literature review on quantitative type theory(§3.1) and typed assembly languages(§3.2), explaining why QTT

is the best approach for runtime erasure and why previous typed assembly languages are not suitable for type-preserving compilation with dependent types. §4 reports my initial progress: the defunctionalization transformation for dependent types (§4.1), a dependently typed assembly language derived from the defunctionalized intermediate representation (§4.2), and defunctionalization for quantitative type theory (§4.3). In §5, I outline the remaining steps toward a QTAL, a quantitatively and dependently typed assembly language, and a timeplan is given at the end.

2 Prerequisites

2.1 Dependent type theory

Dependent type theory is similar to the simply-typed lambda calculus (STLC), presented with the usual typing judgement $\Gamma \vdash M : A$ which means "in typing context Γ, the expression M has type A", where Γ is a list of variables and their associated types. The type theory I informally introduce here is the Calculus of Constructions with predicative universes (also known as CC_ω, see more in e.g. [31, 5, 27]), extended with common datatypes like natural numbers and identity types. Its full syntax can be found in Fig. 2.

$$
\begin{array}{llll}
\text{Expressions} & A, B, L, M, N, P & ::= & x \mid U \mid \Pi x{:}A.\, B \mid M\ N \mid \lambda x{:}A.\, M \\
& & & \mid \Sigma x{:}A.\, B \mid (M, N) \mid \mathbf{let}_P\ (x, y) = M\ \mathbf{in}\ N \\
& & & \mid \mathbf{Nat} \mid \mathbf{zero} \mid \mathbf{suc}\ M \mid \mathbf{iter}_P(M, N, N') \\
& & & \mid \mathbf{Id}_A(M, N) \mid \mathbf{refl}_A(M) \mid \mathbf{J}_C(M, M', P, N) \\
\text{Universes} & U & ::= & U_i \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A
\end{array}
$$

Figure 2: Syntax of CC_ω

We start with the familiar rules for introducing and eliminating lambda abstractions in the simply typed setting.

$$
\textsc{stlc-Lambda}\ \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \to B}
\qquad
\textsc{stlc-Apply}\ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}
$$

Dependent types generalize these rules to express the type dependency of output types on input values. Rule TY-Lambda constructs a dependent function type, or a Π-type, instead of the usual arrow type. It means that the output type B depends on the value of the input (of type A), represented by the variable x (which is bounded in B), since the input value is unknown until the function is applied to an argument. For rule TY-Apply, we substitute the input expression N for x in B to get the result type of the application. As a convention, I write $A \to B$ for non-dependent functions.

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

The most distinguished feature of dependent type theory is the no-distinction between types and terms. In other words, any type can be a treated as a term, and functions are allowed

to take types and return types. Polymorphic functions are just functions with type arguments, for example, the polymorphic identity function is $(\lambda A{:}U_0.\, \lambda x{:}A.\, x)$. Here, U_0 is a universe (also known as a *sort* or a *kind*), which is the type of types. An impredicative universe is a single sort that is the type of all types, including itself. It is simple, yet the impredicativity can be exploited to form a paradoxical construction, breaking the logical consistency of the type theory [16]. Instead, we use a set of universes indexed by natural numbers $\{U_i\}_{i \in \mathbb{N}}$ with the type of U_i being U_{i+1} (a predicative hierarchy), which prevents such paradox.

Since types are terms, $\Gamma \vdash A : U_i$ (for some i) is equivalent to stating that "A is a well-formed type under context Γ" [1]. Ground types like natural numbers and booleans are in U_0. A dependent function $\Pi x{:}A.\, B$ is in the larger universe between the universes of its input type A and its output type B.

$$\textsc{cc-Nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Nat} : U_0} \qquad \textsc{cc-Pi}\ \frac{\Gamma \vdash A : U_i \quad \Gamma, x{:}A \vdash B : U_j}{\Gamma \vdash \Pi x{:}A.\, B : U_{max(i,j)}}$$

A context is well-formed (denoted as $\vdash \Gamma$) when all of its types are well-formed. The variable rule, introductions of base types like **Nat**, and the universe rule (the base cases of type judgements) are only valid under well-formed contexts.

$$\textsc{cc-wf-Empty}\ \frac{}{\vdash \cdot} \qquad \textsc{cc-wf-Cons}\ \frac{\vdash \Gamma \quad \Gamma \vdash A : U}{\vdash \Gamma, x{:}A} \qquad \textsc{cc-Var}\ \frac{x : A \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash x : A} \qquad \textsc{cc-Universe}\ \frac{\vdash \Gamma}{\Gamma \vdash U_i : U_{i+1}}$$

Dependent type theory has the usual $\beta\eta$-equality as in STLC, treated as the *definitional equality* of the type theory, denoted as $\vdash M \equiv N$ [2]. Note that the equivalence also applies to types, for example, $\vdash (\lambda a : U_0.\, a)\ A \equiv A$, by β-reduction. So, two types could be equivalent while being syntactically different, and we have a conversion rule for transporting equivalent types in typing judgements.

$$\textsc{cc-Equiv}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : U \quad \vdash A \equiv B}{\Gamma \vdash M : B}$$

Dependent pairs

In STLC, it is useful to define a product type for pairs of data.

$$\textsc{stlc-Pair}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \textsc{stlc-PairElim}\ \frac{\Gamma \vdash M : A \times B \quad \Gamma, x{:}A, y{:}B \vdash N : P}{\Gamma \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : P}$$

Its elimination rule states that to use an expression of type $A \times B$, it is sufficient to specify the computations on its two canonical components. We get the usual projections $\pi_1\, M$ by $\mathbf{let}\ (x, y) = M\ \mathbf{in}\ x$ and similarly for π_2. As expected, the definitional equality for the eliminator

[1] The context Γ is involved here because the type may depend on variables in it, which makes contexts of dependent types *telescopes* instead of simple lists.

[2] Definitional equality can be given in a typed form like $\Gamma \vdash M \equiv N : A$, which is preferred in general, but I give the untyped version here for simplicity.

is

$$\vdash \mathbf{let}\ (x, y) = (M_1, M_2)\ \mathbf{in}\ N \equiv N[M_1/x, M_2/y].$$

Dependent pairs generalize this idea, allowing the type of the second component to depend on the value of the first, constructing a Σ-type. Note that the elimination rule is also generalized, which can produce a term whose type depends on the value of the pair. This is known as *strong elimination*. Some type theory use a *weak elimination* rule which can only eliminate to types that do not depend on the pair.

$$\textsc{cc-Pair}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash (M, N) : \Sigma x{:}A.\, B} \qquad \textsc{cc-PairElim}\ \frac{\Gamma, z{:}\Sigma x{:}A.\, B \vdash P : U_i \quad \Gamma \vdash M : \Sigma x{:}A.\, B \quad \Gamma, x{:}A, y{:}B \vdash N : P[(x, y)/z]}{\Gamma \vdash \mathbf{let}_P\ (x, y) = M\ \mathbf{in}\ N : P[M/z]}$$

Propositions as types

In STLC, we have the Curry-Howard correspondence between type theory and logic, summarized in the table below.

Type theory		Logic	
Types	A	Propositions	A
Terms	$a : A$	Proofs of propositions	A
Functions	$A \to B$	Implications	$A \supset B$
Products	$A \times B$	Conjunctions	$A \wedge B$
Sums	$A + B$	Disjunctions	$A \vee B$

In dependent type theory, a type B that depends on another type A is viewed as a predicate over elements of A. So, a dependent function of type $\Pi x{:}A.\, B$ that gives an instance $B(a)$ for every a of type A corresponds to the universal quantifier $\forall a \in A. B(a)$. A dependent pair (a, b) of type $\Sigma x{:}A.\, B$ is like a piece of evidence $a \in A$ such that $B(a)$ holds, and it corresponds to the existential quantifier $\exists a \in A.\, B(a)$.

Type theory		Logic	
Dependent types B over $a : A$		Predicates B over $a \in A$	
Dependent functions	$\Pi x{:}A.\, B$	Universal quantifiers	$\forall a \in A.\, B(a)$
Dependent pairs	$\Sigma x{:}A.\, B$	Existential quantifiers	$\exists a \in A.\, B(a)$

We reason about our programs' properties with types, therefore, it is essential that the type theory's corresponding logic is consistent and type checking is decidable, otherwise, our reasoning is unsound or cannot be verified. Since types could be complicated expressions and rule CC-Equiv involves equivalence checking of two types (usually by normalizing them and compare syntactically), the type theory must be terminating for decidable type checking. Termination and consistency of CC_ω are well-established in previous work (e.g. see [31, 18, 44]).

Natural numbers

The constructors of natural numbers are exactly the same as in simply typed languages.

$$\textsc{cc-Zero}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{zero} : \mathbf{Nat}} \qquad \textsc{cc-Suc}\ \frac{\Gamma \vdash M : \mathbf{Nat}}{\Gamma \vdash \mathbf{suc}\ M : \mathbf{Nat}}$$

Its eliminator, an iterator, has a generalized type as it depends on the value it is iterating upon. Logically, this corresponds to the induction principle on natural numbers.

$$\textsc{cc-NatElim}\ \frac{\Gamma, z{:}\mathbf{Nat} \vdash P : U \quad \Gamma \vdash M : \mathbf{Nat} \quad \Gamma \vdash N : P[\mathbf{zero}/z] \quad \Gamma, z{:}\mathbf{Nat}, p{:}P \vdash N' : P[(\mathbf{suc}\ z)/z]}{\Gamma \vdash \mathbf{iter}_P(M, N, N') : P[M/z]}$$

$$\vdash \mathbf{iter}_P(\mathbf{zero}, N, N') \equiv N$$

$$\vdash \mathbf{iter}_P(\mathbf{suc}\ M, N, N') \equiv N'[M/z, \mathbf{iter}_P(M, N, N')/p]$$

Identity types

The definitional equality $\vdash M \equiv N$ is a meta-theoretical notion, which cannot be referred to in our type theory's expressions, so, we cannot use it to reason about equality of our programs. Instead, we introduce *propositional equality*, a datatype whose inhabitants are proofs of two terms being equal.

$$\textsc{cc-Id}\ \frac{\Gamma \vdash A : U_i \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \mathbf{Id}_A(M, N) : U_i} \qquad \textsc{cc-Refl}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{refl}_A(M) : \mathbf{Id}_A(M, M)}$$

It has one constructor, **refl**, and states that only definitionally equal expressions are propositionally equal. This kind of construction is called *intentional equality*. Some variants of dependent type theory uses *extensional equality* that allows propositional equality to imply definitional ones, which makes type checking undecidable.

$$\textsc{cc-ElimId}\ \frac{\begin{array}{c}\Gamma, x{:}A, y{:}A, p{:}\mathbf{Id}_A(x, y) \vdash C : U_i \\ \Gamma, z{:}A \vdash N : C[z/x, z/y, \mathbf{refl}_A(z)/p] \\ \Gamma \vdash M_1 : A \quad \Gamma \vdash M_2 : A \\ \Gamma \vdash P : \mathbf{Id}_A(M, N)\end{array}}{\Gamma \vdash \mathbf{J}_C(M_1, M_2, P, N) : C[M_1/x, M_2/y, P/p]}$$

Identity type's elimination rule is also known as the J-rule, and the usual properties of equivalence relations like symmetry, transitivity, and the Leibniz rule are all special cases of it (for more examples, see [27, 39]). It seems scary, but it follows the general principle of eliminators: for each constructor, it specifies the corresponding computation, and eliminates to a type that depends on the value to be eliminated. In this case, **refl** is the only constructor, P is a proof of M_1 equals to M_2, and N is the computation to be performed on M_1 (or equivalently, M_2), as its definitional equality suggests.

$$\vdash \mathbf{J}_C(M, M, \mathbf{refl}_A(M), N) \equiv N[M/z]$$

Inductive families

The general way of defining datatypes like the Σ-type, natural numbers, and identity types above is inductive families, a dependently typed generalization of algebraic datatypes. Inductive families are conceptually straightforward but technically involved due to their generality, so, I will not cover the type-theoretic notations (see e.g. [44, 15]) but instead convey the ideas and show how to define them in Agda.

To specify an inductive type, we must state the types of its parameters and indices, and which universe the inductive definition lives in. Then, we give it a set of constructors, each takes a list of arguments and returns an member of the datatype with arbitrary instantiated indices. For example, the following definition of vectors has a type parameter A (written before the semicolon) and a natural number index n.

```
data Vec (A : Set₀) : (n : Nat) → Set₀ where
  [] : Vec A 0
  Cons : ∀{n} → A → Vec A n → Vec A (suc n)
```

Set_0 is U_0 in Agda. The ∀-operator in Cons states that n is an *implicit argument*, which can be automatically inferred by the compiler (since it is completely determined by the type of the list we are consing) and saves the user from the syntactic burden of writing it explicitly.

Inductive definitions are subjected two restrictions to ensure consistency of the type theory. First, the *universe checking* rejects impredicative definitions, such as constructors that take elements in a larger universe than the universe of the datatype. Second, the *positivity checking* rejects any reference to the datatype itself at positions other than strictly-positive ones, i.e. self-reference cannot appear at the left side of an arrow or inside its own type indices.

Inductive families are eliminated with pattern matching functions, and Agda uses a syntactic condition to ensure termination. It only accepts recursive definitions that could be assigned with a lexicographical order on its arguments such that the arguments are decreasing with each recursive call.

2.2 Compilation of dependent types

Fig. 3 gives a simplified and abstract view of current compilers for dependently typed languages. We start with a surface-level user language that contains syntactic sugar and implicit arguments. The first step is desugaring, then, an *elaboration stage* infers the implicit arguments and turns the user language into a *core calculus* where everything is explicit, much like the CC_ω that we saw previously. Once the core-calculus expressions are type-checked, the compiler erases their types and performs the conventional transformations and optimizations, generating untyped assembly code and eventually, binary executable.

Bidirectional checking

Type checking is implemented with a bidirectional algorithm that separates the typing rules into two modes: checking ($\Gamma \vdash M \Leftarrow A$) and synthesizing ($\Gamma \vdash M \Rightarrow A$). Checking takes (Γ, M, A) as inputs and checks if the expression M can be typed with A. Synthesizing takes (Γ, M) as inputs and outputs the inferred type of M if it is well-typed, and fails otherwise. For dependent types, the only checking rule is rule CC-Equiv, and all other rules are synthesizing.

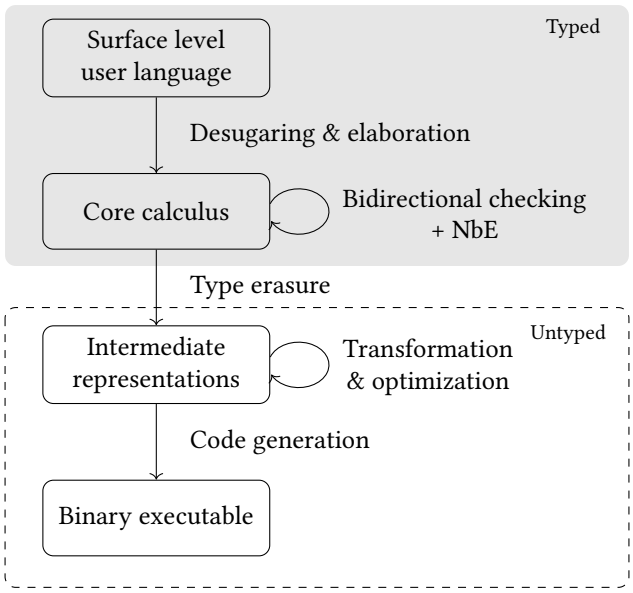


Figure 3: Structure of existing compilers for dependently typed languages

$$\textsc{check-Equiv}\quad \frac{\Gamma \vdash M \Rightarrow A' \qquad \vdash A \equiv A'}{\Gamma \vdash M \Leftarrow A}$$

In other words, to check if M has type A under Γ, we infer the type of M under Γ and see if the result is equivalent to A.

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

The inferred type of a lambda abstraction is $\Pi x{:}A.\, B$, where A is already available and B is inferred from the function body. To infer the type of an application $M\ N$, we first infer the type of M. If it is a function and N matches the function's input type, we conclude that the inferred type of the application is $B[N/x]$; in other cases, the algorithm fails as the application is ill-typed. Types of variables are inferred directly from the context, and the rules for Π-types and universes are straightforward.

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

Before, it was unclear when to use the rule cc-Equiv in a type derivation tree, as it seems to fit anywhere. Bidirectional typing rules clarify the order of which rule should be applied in a derivation, turning the type theory into a practical type checking algorithm. Bidirectional checking for dependent types is developed by Coquand [17], and a survey on various bidirectional type systems is available at [22].

Normalization by evaluation

Equivalence test for terms is essential in type checking. Dependent type theories like CC_ω are strongly normalizing, so, equivalence checking is simply implemented as a comparison between the normal forms. It is important to have an efficient normalization algorithm, as equivalence tests often involve normalizing long and complicated expressions in practice. The conventional approach of finding β-redices and doing substitutions to get β-short normal forms is not suitable for this task — substitution is a costly operation that drastically slows normalization down.

Normalization by evaluation (NbE) is the more efficient alternative. Instead of performing syntactic substitutions, it interprets the code in a semantic domain, where complicated expressions can be quickly evaluated to semantic values, and then *reified* back to the syntax as normal forms. The semantic domain supports the evaluation of open terms (similar to partial evaluation [20]) because neutral terms, which are irreducible expressions stucked on variables, are *reflected* syntactically into the domain.

In practice, our semantic objects are the runtime representations of programs under a runtime environment that maps the free variables to their reflected semantic values. Normal forms of the base types are included syntactically. Lambda abstractions are interpreted as closures, which are the syntax of functions paired with their current runtime environment. To evaluate the application of a closure on a semantic value, we simply interpret the syntactic function body in the runtime environment with a new entry that associates the function's bound variable to the input value. To evaluate a variable, we just look it up from the runtime environment, and no substitution is required.

The simply-typed NbE was discovered by Berger and Schwichtenberg [7], and Abel provides a comprehensive analysis on NbE for dependent types [1]. NbE is deeply connected with categorical semantics, for example, it amounts to the categorical gluing of presheaves [23, 3].

3 Literature review

In this section, I give a literature review on the most relevant previous work: quantitative type theory(§3.1) and typed assembly languages(§3.2). I explain why QTT is superior to graded type theory and whole-program analysis for runtime erasure, and why no existing typed assembly language is suitable for type-preserving compilation with dependent types.

3.1 Quantitative type theory

Quantitative type theory (QTT) combines dependent type theory with linearity [4, 32], which marks the usage of computational resources such as variables in the context and inputs of functions. The type judgement of QTT is in the following form (with the context Γ expanded):

$$x_1 \overset{q_1}{:} A_1,\ \ldots,\ x_n \overset{q_n}{:} A_n \vdash M \overset{\sigma}{:} A$$

Each variable is tagged with a quantity that ranges from 0 (never used), 1 (used once), and ω (used many times). The addition and multiplication defines how usages can be combined.

$+$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$\cdot$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

Addition and multiplication extends to contexts. A multiplication $q\Gamma$ is multiplying the quantity of each variable in Γ by q. So, $0\Gamma_1 = 0\Gamma_2$ implies that Γ_1 and Γ_2 contain the same variable-type pairs, with possibly different usage annotations. Addition for contexts is pointwise, and $\Gamma_1 + \Gamma_2$ is only well-defined when $0\Gamma_1 = 0\Gamma_2$.

The type judgement is tagged with $\sigma \in \{0, 1\}$. When $\sigma = 0$, we construct a term M in the *erased fragment* which is runtime irrelevant; when $\sigma = 1$, M is in the *present fragment* which will be used at runtime (no matter how many times). We get the typing rules of QTT by adding quantity annotations to the rules of CC_ω with a principle: zero needs nothing. If $\Gamma \vdash M \overset{0}{:} A$, then all variables in the context must have quantities 0 (i.e. $\Gamma = 0\Gamma$). Since types are always computationally irrelevant, they are judged with $\sigma = 0$ and the contexts are multiplied by zero to enforce the zero-needs-nothing property.

$$\textsc{qtt-Pi}\ \frac{0\Gamma \vdash A \overset{0}{:} U_i \qquad 0\Gamma, x \overset{0}{:} A \vdash B \overset{0}{:} U_j}{0\Gamma \vdash \Pi x \overset{q}{:} A.\, B \overset{0}{:} U_{max(i,j)}} \qquad \textsc{qtt-Universe}\ \frac{\vdash 0\Gamma}{0\Gamma \vdash U_i \overset{0}{:} U_{i+1}}$$

As you can see, the dependent function type associates a quantity q to its bound variable x that records how many times the input will be used. Rule QTT-Lambda is judged with the premise that the bound variable x is used for $q\sigma$ times – we multiply q by σ to comply with the zero-needs-nothing principle (the usage of x is 0 when $\sigma = 0$ and q when $\sigma = 1$). Intuitively, if a term M uses an variable for m times to construct a function of type $\Pi x \overset{q}{:} A.\, B$, and another term N (of type A) uses the same variable for n times, then, the total usage of that variable in the application $M\ N$ is $m + qn$ times. So, the application rule is judged under $\Gamma_1 + q\Gamma_2$, correctly summing the resource usage in the contexts. Its constraint in the premises, again, is enforcing the zero-needs-nothing property by stating that if the function's argument is runtime irrelevant ($\sigma' = 0$), then either the whole application is irrelevant ($\sigma = 0$) or the function never uses its input ($q = 0$).

$$\textsc{qtt-Lambda}\ \frac{\Gamma, x \overset{q\sigma}{:} A \vdash M \overset{\sigma}{:} B}{\Gamma \vdash \lambda x \overset{q}{:} A.\, M \overset{\sigma}{:} \Pi x \overset{q}{:} A.\, B} \qquad \textsc{qtt-Apply}\ \frac{0\Gamma_1 = 0\Gamma_2 \qquad \sigma' = 0 \Leftrightarrow (q = 0 \vee \sigma = 0) \qquad \Gamma_1 \vdash M \overset{\sigma}{:} \Pi x \overset{q}{:} A.\, B \qquad \Gamma_2 \vdash N \overset{\sigma'}{:} A}{\Gamma_1 + q\Gamma_2 \vdash M\ N \overset{\sigma}{:} B[N/x]}$$

The variable rule is straightforward: either x is used once in the present fragment, or it is used zero times in the erased fragment. In the conversion rule, we enforce that B is constructed without resources, and everything else remains the same as before [3].

$$\textsc{qtt-Var}\ \frac{\vdash 0\Gamma, x \overset{\sigma}{:} A, 0\Gamma'}{0\Gamma, x \overset{\sigma}{:} A, 0\Gamma' \vdash x \overset{\sigma}{:} A} \qquad \textsc{qtt-Equiv}\ \frac{\Gamma \vdash M \overset{\sigma}{:} A \qquad 0\Gamma \vdash B \overset{0}{:} U \qquad \vdash A \equiv B}{\Gamma \vdash M \overset{\sigma}{:} B}$$

In general, we can pick quantities from any semiring $(Q, 0, 1, +, \cdot)$ that is *positive* and has the *zero-product property*. A semiring is a set Q with binary operators $(+)$ and $(\cdot)$, and two elements 0 and 1 such that $(Q, 0, +)$ is a commutative monoid, $(Q, 1, \cdot)$ is a monoid, $(+)$ and $(\cdot)$ distribute, and $0 \cdot q = 0 = q \cdot 0$ for all $q \in Q$. In a positive semiring, $q_1 + q_2 = 0$ implies $q_1 = 0$

[3] The premise should be $0\Gamma \vdash A \equiv B \overset{0}{:} U$ if we are using typed equality.

and $q_2 = 0$; the zero-product property means that $q_1 \cdot q_2 = 0$ implies $q_1 = 0$ or $q_2 = 0$. These properties are essential for substitutions to be well-typed.

Among other approaches to runtime erasure, QTT is the most suitable one for type-preserving compilation. Compared with erasure based on whole-program analysis [13, 43] which fails to infer erasable function arguments in some cases, QTT allows users to mark precisely which argument should be erased, and has better type-checking performance [12]. It will not add syntactic burden to the users, because most of the quantity annotations can be automatically inferred, for example, the implicit arguments have usage 0 (since they only appear in types). The compiler can easily identify erasable function closures in QTT – the erased functions are judged with $\sigma = 0$ and the runtime relevant functions are judged with $\sigma = 1$. Graded modal type theory, a generalized version of QTT, discards the "erased" and "present" fragments and replaces them with a *modal type* $\Box_q A$ for runtime usage of expressions. $M : \Box_q A$ means that M is a piece of code that will be used q times during execution (q comes from an algebra of quantities, such as a semiring). [4] Graded modal type theory supports more definitional equality than QTT [14, 34], however, it makes erasing function closures difficult, as it is no longer apparent whether a lambda abstraction of type $\Pi x \overset{q}{:} A.\, B$ is used at the runtime or not.

3.2 Typed assembly languages

Typed assembly languages were inspired by TIL, a type-directed optimizing compiler for ML [42]. TIL was based on a key observation: the more expressive a programming language's type system is, the more useful these types are to compilers. TIL used type information to generate efficient specialized code for different instances of a polymorphic function, speed up garbage collection, and perform conventional optimizations reliably. It transforms the source program through a chain of typed intermediate languages, preserving and utilizing the types along the way, and eventually discards the types during target code generation, since there was no place for them in the usual untyped assembly.

TAL

The idea of having a typed assembly language (TAL) [36] follows naturally from TIL, and completes a compiler that preserves the type information from source to target. Here is a TAL program that takes an input and returns a pair of its copies.

```
copy : code['a]{r1 : 'a, r2 : {r1 : <'a , 'a>}}.
  malloc r3, <r1 , r1>
  mov r1 r3
  jmp r2
```

Without the type annotation, it is the same as an assembly language for a register machine with a small RISC-style instructions set. Instruction `malloc` creates a pair of copies of `r1`'s content on the heap and stores its address in `r3`. Then, this address is moved to `r1` and the program jumps to the return address in `r2`. Its type specifies that `copy` is a code block which is polymorphic over some type α, and it requires `r1` to contain a value of type α and `r2` to contain the address to a block that expects a pair of α-typed values in `r1`. The type checker ensures that the pre-condition is always satisfied when the control flow transfers to the start of a block, and there is no ill-typed operations (like adding an integer to an abstract type).

[4] In [34], the modal type is given as a part of the syntax; in [14], $\Box_q A$ is given as an encoding by the dependent pair $\Sigma x \overset{q}{:} A.Unit$.

TAL code is generated from System F expressions with a series of type-preserving transformations. The compiler first uses type-preserving CPS transformation and closure conversion to expose control flows and function closures, then turns tuple constructors into explicit allocations, which makes the resulting code very similar to TAL, and finishes with a straightforward code generation. Type signatures are removed before binary-code generation and execution. Typed closure conversion produces closure objects in existential types that can be packed or unpacked, which are all supported in TAL. The generated compiler output could be further optimized, for example, removing dead code, unnecessary closures, and redundant move instructions.

In practice, Morisset et al. developed TALx86, a strongly typed fragment of the the Intel IA32 assembly language [19]. It uses a stack-based computational model with returning functions, but supports CPS-based compilation by providing mechanisms to perform direct jumps.

DTAL

Xi and Harper extended TAL with a limited form of dependent types, giving a dependently typed assembly language (DTAL) [48]. It provides two dependent datatypes, both indexed over integers: `int(x)` whose only inhabitant is the integer `x`, and polymorphic arrays `'a array(x)` with lengths of `x`. A code block's type signature can universally quantify over these integer indices and add constraints to them. The constraints are given in a special syntax of arithmetic and propositional logic, separating the assembly code and the reasoning, preventing ambiguous scenarios like "a type that depends on the content of a mutable register".

The DTAL code below specifies a block that takes an α-array of length `m` in `r1`, an integer `n` in `r2`, and a value of type α in `r3` (for all types α and integers `(m, n)`). It further constraints that `m >= 0`, `n >= 0`, and `n < m`. The code block uses `load` to copy the value in `r3` into the array's position `n`.

```
arraySet : ('a){m : int | m >= 0 , n : int | n >= 0 && n < m}
      [r1 : 'a array(m), r2 : int(n), r3 : 'a]
      load r1(r2), r3
      halt
```

As in TAL, the type checker ensures that all registers contain values of the specified types when control enters the block and all operations are well-typed. In addition, it uses an SMT solver to check if all constraints are satisfied before the control enters a code block. Inside these blocks, it assumes the constraints in the pre-conditions holds, and continuously updates them while sequentially walking through each instruction, checking for out-of-bound array access. In `arraySet`, the constraints before loading comes directly from the pre-conditions, which implies that `m` and `n` are both greater than 0 and `n < m`, so, the loading is safe. If we put `add r2, r2, 1` before `load`, then, `n < m` is no longer true in the updated constraints after this instruction, which makes the loading possibly unsafe, and the code is rejected.

Like TAL, type signatures of DTAL code blocks are erased before binary-code generation and execution. The logical values `m` and `n` plays no role in computation (since they can only appear in the type signature), so, erasure will not change program behavior and we do not need to store the logical values at runtime. Since well-typed DTAL code promises no out-of-bound access, runtime array bound checks can be safely eliminated as an optimization.

DTAL is suitable for compiling early dependently typed languages like Dependent ML and Xanadu that features a limited form of dependent types similar to the types in DTAL. Its design choice of separating the reasoning logic and the assembly language is worth learning from,

despite its reliance on an external SMT solver and the lack of the full dependent type theory.

Singleton

Winwood presented a design of a general purpose dependently typed assembly language, Singleton [47]. Its style is similar to TAL, but it supports full dependent types, inductive families, and a `case` branch instruction for pattern matching. It features the use of singleton types $sgl(a : A)$ whose only inhabitant is the value a of type A [5] , like the generalized version of `int(x)` as we saw in DTAL. Upon the assembly instructions, Singleton has an assertion logic based on the Calculus of Inductive Constructions for specifying types of registers and reasoning about their contents. Here is the type of a code block that takes the head of a list (taken from [47]).

```
head: ∀(A : Set)(xs : list A).
      {a1 : sgl (xs : list A),
       ra : ∀(x : A)(xs' : list A)(pf : xs = Cons x xs').
            {t1 : sgl(x : A)}}
```

The code block takes two *logical arguments* A and xs, which are values only relevant to type checking and can be erased at runtime. These arguments are not automatically inferred – they need to be provided to the code block in the assembly. The `head` block expects `a1` to contain exactly the list xs (as specified by the singleton type), and the return address `ra` to point to a code block which requires three logical values `x`, `xs'`, and `pf` such that `pf` proves that `xs = Cons x xs'` and `x` is stored in `t1`. Singleton's type system guarantees that if `head` returns from `ra`, then it correctly implements a head function. However, this correctness guarantee is partial since `head` could throw an exception (when `xs` is empty, for example), return from another address or loop forever.

Unfortunately, Singleton is incompatible with the type-preserving compiler transformations for dependent types, which are usually different from the non-dependent versions. For example, Singleton has existential types to accommodate function closures, yet closure conversion with existential is not type-preserving in dependent settings [10]. There is also no way for erasing computationally irrelevant type indices, so, a Singleton implementation of vectors still has to pay extra space and time to store and compute their lengths.

Discussion

The following table summarizes the features of the typed assembly languages reviewed in this section. We can see that there does not exist a suitable assembly language which supports full dependent types, allows precise erasure of runtime irrelevant computations and type indices, and is compatible with the dependently typed transformations. Therefore, the development toward type-preserving compilers for dependent types is stuck, and it could only proceed when a typed assembly language with these properties becomes available.

Language	Fully dependent types	Erasure	Type-preserving compilation
TAL	✗	✓	✓
DTAL	✗	✓	✓
Singleton	✓	✗	✗

[5] The syntax might be confusing: a and A are really type indices here and sgl is not a binding operator!

4 Progress report

In this section, I describe the initial work done toward the thesis. It includes the defunctionalization transformation for dependent types (§4.1), a dependently typed assembly language derived from the defunctionalized intermediate representation (§4.2), and defunctionalization for quantitative type theory (§4.3).

In parallel, I completed a mechanized proof of the correctness of normalization by hereditary substitution for STLC in Agda (§4.4). Its generalization in contextual modal type theory is related to the notion of my defunctionalized labels, and the subject is worth studying in its own right.

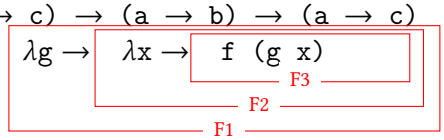
4.1 Defunctionalization with dependent types

The *defunctionalization* translation turns higher-order programs into first-order programs. Previous work on defunctionalization in typed settings has examined a variety of languages, from simply-typed [38] and monomorphizable [6] to fully polymorphic [41], but not dependently typed ones. This section introduces the traditional typed defunctionalization for polymorphic languages (as in [41]), explains why such approach fails to generalize to dependent types, and provides my solution – a dependently typed calculus for defunctionalized intermediate representations. The contents presented here is adapted from my published paper on PLDI (see [28]), co-authored with my supervisor, and part of the work on DCC was completed before the start of my PhD.

Defunctionalization with polymorphic types

Defunctionalization for polymorphic languages replaces the function arrow `_→ _` with a first-order data type `_⇝ _`. Here is an example that defunctionalizes the polymorphic `compose` function which contains three abstractions, here labeled F1, F2, and F3.

```
compose :: (b → c) → (a → b) → (a → c)
compose = λf → λg → λx → f (g x)
```



Defunctionalizing `compose` produces a data type `_⇝ _` with one constructor for each abstraction, and the constructor's arguments correspond to the free variables in the abstraction. `F2` has one argument of type `b ⇝ c`, corresponding to `f` in F2 above, and `F3` has two arguments, corresponding to `f` and `g` in F3.

```
data (⇝) a b where
  F1 :: (b ⇝ c) ⇝ (a ⇝ b) ⇝ (a ⇝ c)
  F2 :: (b ⇝ c) → (a ⇝ b) ⇝ (a ⇝ c)
  F3 :: (b ⇝ c) → (a ⇝ b) → (a ⇝ c)
```

Defunctionalization also produces an operator `$` that maps the constructors of `_⇝ _` to the bodies of the corresponding abstractions:

```
($) :: (a ⇝ b) → a → b
F1 $ f = F2 f
F2 f $ g = F3 f g
F3 f g $ x = f $ (g $ x)
```

Here `$` maps `F1` and the argument x to `F2 x`, since the body of the abstraction F1 is F2, with `x` free. Similarly, it maps `F3` to `f $ (g $ x)` because the body of F3 is `f (g x)`.

Finally, defunctionalization replaces $\to$ with $\rightsquigarrow$ and F1 with `F1` in `compose` itself, and the program contains no higher order function anymore.

```
compose_ :: (b ⇝ c) ⇝ (a ⇝ b) ⇝ (a ⇝ c)
compose_ = F1
```

In general, defunctionalization replaces each abstraction $\lambda x.M_i$ in the source program with a constructor application $C_i\ \overline{y}$ where C_i is a constructor of $\_ \rightsquigarrow \_$ and $\overline{y}$ are the free variables of the abstraction, and replaces each application $f\ x$ with $f\ \$\ x$, where the infix operator $\$ maps C_i back to M_i. The data type $\_ \rightsquigarrow \_$ produced by defunctionalization is a *generalized algebraic data type* (GADT), in which the return type of each constructor can have a distinct instantiation of the type parameters, and constructor types can involve type variables (such as `b` in the type of `F3`) that do not appear in return types.

Problems of using inductive families

Polymorphic functions abstract over types and defunctionalize to GADTs indexed by types. Despite the analogy, dependent functions which abstract over expressions cannot defunctionalize to inductive families indexed by expressions, due to restrictions imposed on inductive families to ensure consistency (see §2.1).

$$\begin{array}{ll}
\mathsf{data}\ _\rightsquigarrow_ : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} & _\$_ : \forall\ \{A\ B\} \to (A \rightsquigarrow B) \to A \to B \\
\quad \mathsf{F1} : (B \rightsquigarrow C) \rightsquigarrow (A \rightsquigarrow B) \rightsquigarrow (A \rightsquigarrow C) & \mathsf{F1}\ \$\ f = \mathsf{F2}\ f \\
\quad \mathsf{F2} : (B \rightsquigarrow C) \to (A \rightsquigarrow B) \rightsquigarrow (A \rightsquigarrow C) & \mathsf{F2}\ f\ \$\ g = \mathsf{F3}\ f\ g \\
\quad \mathsf{F3} : (B \rightsquigarrow C) \to (A \rightsquigarrow B) \to (A \rightsquigarrow C) & \mathsf{F3}\ f\ g\ \$\ x = f\ \$\ (g\ \$\ x)
\end{array}$$

For example, we can easily write the defunctionalized `compose` above in Agda with inductive families, but it is rejected since the constructor F1 takes an argument of type $B \rightsquigarrow C$ that inhabits the universe Set_1 (which is larger than its universe Set), and in the type of F1, $\rightsquigarrow$ is indexed by $\rightsquigarrow$ itself, so the definition fails positivity checking. In more complicated scenarios where dependent functions are involved, the definition of $ might fail Agda's termination checking.

The fact that Agda rejects the inductive families generated by defunctionalization suggests that inductive families are ill-suited to the task. For example, the universe restriction that rejects the constructors of $\rightsquigarrow$ does not apply to the closures that correspond to those constructors in the source program: there is nothing requiring a free variable in an abstraction body to inhabit a smaller universe than the function itself. The additional restriction arises from an expressivity mismatch: the universe restriction is only needed when inductive families are not used in a closure-like fashion — e.g. when constructor arguments are extracted.

A defunctionalized Calculus of Constructions

Instead of using inductive families, I choose to follow the direction of *abstract transformation*, which studies transformations into a specialized target language with new constructs for defunctionalized datatypes. Transforming into these constructs captures the essence of the translation, while avoiding the unnecessary restrictions imposed by more concrete settings. A similar approach is used in the study of abstract closure conversion [33, 10].

The target language I define is the Defunctionalized Calculus of Constructions (DCC), similar to the CC_ω introduced in Section 2, but with a new construct for defunctionalized *labels*

$$
\begin{array}{rcll}
\mathfrak{D} & ::= & \mathfrak{L}_3(\{f : B \to C, g : A \to B\}, x : A & \mapsto \quad f\ @\ (g\ @\ x) : C), \\
 & & \mathfrak{L}_2(\{f : B \to C\}, g : (A \to B) & \mapsto \quad \mathfrak{L}_3\{f, g\} : A \to C), \\
 & & \mathfrak{L}_1(\{\}, f : (B \to C) & \mapsto \quad \mathfrak{L}_2\{f\} : (A \to B) \to A \to C) \\
\text{compose} & ::= & \mathfrak{L}_1\{\} &
\end{array}
$$

Figure 4: Defunctionalized simply-typed composition

in place of lambda abstractions. Fig. 4 shows the result of defunctionalizing the simply-typed compose function

$$\lambda^1 f{:}(B \to C).\, \lambda^2 g{:}(A \to B).\, \lambda^3 x{:}A.\, f\ (g\ x)$$

to DCC [6], which looks and behaves like the conventional defunctionalization presented previously. In our translation into DCC, each abstraction $\lambda x.M_i$ is replaced with a *label expression* $\mathfrak{L}_i\{\overline{y}\}$ where $\mathfrak{L}_i$ is the label's identifier and $\overline{y}$ are the abstraction's free variables. The function body M_i is stored in a separate *label context* $\mathfrak{D}$ indexed by the label identifier, along with its typing information.

In Fig. 4, the label context $\mathfrak{D}$ has three entries, one for each abstraction in the original `compose` function. Each entry corresponds to one case of the `$` function in the conventional defunctionalization. For example, $\mathfrak{L}_3$ arises from the translation of $\lambda x : A.\ f\ (g\ x)$, and corresponds to the `F3` case in the definition of `$`: it has two free variables $f : B \to C$ and $g : A \to B$, a bound variable x, and a body $f\ @\ (g\ @\ x)$. As we shall see, a label application $\mathfrak{L}_3\{f, g\}\ @\ N$ reduces to $f\ @\ (g\ @\ N)$, just as the application `(F3 f g) $ x` reduces to the corresponding right hand side `f $ (g $ x)`.

Syntax and meta-theory of DCC

The syntax of DCC shown below is given in a different colour and font for distinction. Its expressions are similar to that of CC_ω, except that DCC contains first-class function labels $\mathfrak{L}\{\overline{M}\}$ instead of lambda abstractions.

$$
\begin{array}{lllll}
\text{Universes} & U & ::= & U_i & \\
\text{Expressions} & A, B, L, M, N & ::= & x \mid U \mid \Pi x{:}A.B \mid L\ @\ M \mid \mathfrak{L}\{\overline{M}\} & \\
\text{Type contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A & \\
\text{Label contexts} & \mathfrak{D} & ::= & \cdot \mid \mathfrak{D}, \mathfrak{L}(\{\overline{x}{:}\overline{A}\}, x{:}A \mapsto M : B) & \\
\text{DCC contexts} & \mathfrak{D};\Gamma & & &
\end{array}
$$

There are two varieties of context in DCC. As in CC, type contexts Γ associate variables x with types A. Label definition contexts $\mathfrak{D}$ pair label names with their associated data: $\mathfrak{L}(\{\overline{x}{:}\overline{A}\}, x{:}A \mapsto M : B)$. Here $\overline{x}{:}\overline{A}$ records the type of the (possibly empty) telescope of free variables that the label takes, $(x : A) \to B$ specifies the label type, and M is the expression to which the label reduces when applied to an argument. Note that types in a type context Γ may refer to labels $\mathfrak{L}_1, \mathfrak{L}_2, \cdots$ in the label context $\mathfrak{D}$, but not vice versa.

The type judgement is in the form of $\mathfrak{D}; \Gamma \vdash M : A$ and well-formedness of the dual contexts are noted as $\vdash \mathfrak{D}; \Gamma$. Rules for variables, universes, Π-types, applications, and conversion are identical to their counterpart rules in CC, like the variable, Π-types, universe, and conversion rules below.

[6] Assuming that A, B, and C are fixed base types here, and each lambda abstraction is tagged with a unique natural number identifier.

$$\textsc{dcc-Var}\ \frac{x : A \in \Gamma \qquad \vdash \mathfrak{D};\Gamma}{\mathfrak{D};\Gamma \vdash x : A} \qquad \textsc{dcc-Universe}\ \frac{\vdash \mathfrak{D};\Gamma}{\mathfrak{D};\Gamma \vdash U_i : U_{i+1}}$$

$$\textsc{dcc-Pi}\ \frac{\mathfrak{D};\Gamma \vdash A : U_i \quad \mathfrak{D};\Gamma, x{:}A \vdash B : U_j}{\mathfrak{D};\Gamma \vdash \Pi x{:}A.B : U_{\max(i,j)}} \qquad \textsc{dcc-Equiv}\ \frac{\mathfrak{D};\Gamma \vdash M : A \quad \mathfrak{D};\Gamma \vdash B : U_i \quad \mathfrak{D} \vdash A \equiv B}{\mathfrak{D};\Gamma \vdash M : B}$$

We focus on the rule for labels. A label term $\mathfrak{L}\{\overline{M}\}$ is well-typed in $\mathfrak{D};\Gamma$ if the following conditions are satisfied.

1. The context $\mathfrak{D};\Gamma$ is *well-formed*.
2. $\mathfrak{L}(\{\overline{x}{:}\overline{A}\}, x{:}A \mapsto M : B)$ is present in $\mathfrak{D}$.
3. The length of the two lists $\overline{M}$ and $\overline{x}{:}\overline{A}$ are equal.
4. All expressions in $\overline{M}$ are well-typed, and their types match the specified types of free variables $\overline{A}$.

Specifically, condition (4) means:

$$\begin{aligned}&\mathfrak{D};\Gamma \vdash M_1 : A_1,\\ &\mathfrak{D};\Gamma \vdash M_2 : A_2[M_1/x_1],\\ &\cdots,\\ &\mathfrak{D};\Gamma \vdash M_n{:}A_n[M_1/x_1, \cdots, M_{n-1}/M_{n-1}].\end{aligned}$$

Each A_{i+1} depends on $x_1, \cdots, x_i$, so $M_1, \cdots, M_i$ need to be substituted in A_{i+1} in the type judgement for M_{i+1}. The type of $\mathfrak{L}\{\overline{M}\}$ is $\Pi x{:}A[\overline{M}/\overline{x}].B[\overline{M}/\overline{x}]$.

Note that values of free variables $\overline{M}$ are substituted in $\Pi x{:}A.B$, the specified type of the label. We use $[\overline{M}/\overline{x}]$ as a syntactic sugar of $[M_1/x_1, \cdots, M_n/x_n]$, and conditions (3) and (4) are abbreviated to $\mathfrak{D};\Gamma \vdash \overline{M} : \overline{A}$ as a convention. Therefore, we have the following rules for labels and applications:

$$\textsc{dcc-Label}\ \frac{\mathfrak{D};\Gamma \vdash \overline{M} : \overline{A} \quad \mathfrak{L}(\{\overline{x}{:}\overline{A}\}, x{:}A \mapsto M : B) \in \mathfrak{D}}{\mathfrak{D};\Gamma \vdash \mathfrak{L}\{\overline{M}\} : \Pi x{:}A[\overline{M}/\overline{x}].B[\overline{M}/\overline{x}]} \qquad \textsc{dcc-Apply}\ \frac{\mathfrak{D};\Gamma \vdash M : \Pi x{:}A.B \quad \mathfrak{D};\Gamma \vdash N : A}{\mathfrak{D};\Gamma \vdash M\ @\ N : B[N/x]}$$

The dual context $\mathfrak{D};\Gamma$ is well-formed when all types in Γ are well-typed (similar to CC_ω), and every label is associated with a well-typed data. In other words, if we have $\mathfrak{L}(\{\overline{x}{:}\overline{A}\}, x{:}A \mapsto M : B)$, M should have the type B as specified in the context formed by the previous label context (it cannot call itself) and the free variables in M (namely $\mathfrak{D};\overline{x}{:}\overline{A}, x{:}A$).

$$\textsc{dcc-wf-Label}\ \frac{\mathfrak{D};\overline{x}{:}\overline{A}, x{:}A \vdash M : B}{\vdash \mathfrak{D}, \mathfrak{L}(\{\overline{x}{:}\overline{A}\}, x{:}A \mapsto M : B);\cdot}$$

When a label term $\mathfrak{L}\{\overline{M}\}$ is applied to an argument N, we can β-reduce it by finding the label's body L in the label context, then substitute the values $\overline{M}$ for its free variables $\overline{x}$ and the

argument N for its bound variable x, as described by the reduction relation $\mathfrak{D} \vdash \mathsf{M} \triangleright \mathsf{N}$.

$$\textsc{dcc-red-Beta}\quad \frac{\mathfrak{L}(\{\overline{\mathsf{x}}{:}\overline{\mathsf{A}}\}, \mathsf{x}{:}\mathsf{A} \mapsto \mathsf{L} : \mathsf{B}) \in \mathfrak{D}}{\mathfrak{D} \vdash \mathfrak{L}\{\overline{\mathsf{M}}\} \mathbin{@} \mathsf{N} \triangleright \mathsf{L}[\overline{\mathsf{M}}/\overline{\mathsf{x}}, \mathsf{N}/\mathsf{x}]}$$

As a dependent type theory, DCC is type-safe and consistent — no terms are non-terminating and no paradox can be derived under the empty typing context.

Theorem 4.1 (Type safety) *If $\mathfrak{D}; \cdot \vdash \mathsf{M} : \mathsf{A}$, then $\mathfrak{D} \vdash \mathsf{M} \triangleright^* \mathsf{v}$ for some irreducible value v.*

Theorem 4.2 (Consistency) *There is no pair of a label context $\mathfrak{D}$ and DCC term M such that $\mathfrak{D}; \cdot \vdash \mathsf{M} : \Pi\mathsf{A}{:}\mathsf{U}.\mathsf{A}$.*

Defunctionalization transformation

The transformation from the source language into DCC needs two processes: a *term transformation* (defined with $\Gamma \vdash M : A \rightsquigarrow \mathsf{M}$) that produces a DCC term from the input term, and a *function extraction* (defined with $\Gamma \vdash M : A \rightsquigarrow_d \mathfrak{D}$) that gives a DCC label context which contains all the function definitions appeared in the input. In other words, the transformation takes the whole derivation tree of a well-typed source language term as input. I write $[\![M]\!]$ for the term transformation and $[\![M]\!]_d$ for the function extraction when Γ and A are obvious.

The term transformation turns lambda abstractions into label terms, using FV to compute the list of free variables. Note that FV is not as simple as finding all the unbound variables in the function body, since the types of these unbound variables might contain other variables, and so on. Instead, FV recursively finds all the variables from the context required to correct type the function. Applications are transformed structurally, and the same applies to variables and types (rules omitted).

$$\textsc{t-Lambda}\quad \frac{FV(\lambda^i x{:}A.M) = \overline{x}{:}\overline{A} \qquad \Gamma \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{\mathsf{x}}}{\Gamma \vdash \lambda^i x{:}A.M : \Pi x{:}A.B \rightsquigarrow \mathfrak{L}_{\mathsf{i}}\{\overline{\mathsf{x}}\}}$$

$$\textsc{t-Apply}\quad \frac{\Gamma \vdash L : \Pi x{:}A.B \rightsquigarrow \mathsf{L} \qquad \Gamma \vdash M : A \rightsquigarrow \mathsf{M}}{\Gamma \vdash L\ M : B[M/x] \rightsquigarrow \mathsf{L} \mathbin{@} \mathsf{M}}$$

We now look at the function extraction process. Rule D-Lambda seems complicated, but it states a simple fact: to extract functions from a lambda abstraction, we work out the transformed terms required to assemble the label definition for this function, and append it after the functions we extracted from M and the argument type A.

$$\textsc{d-Lambda}\quad \frac{\begin{array}{c}\Gamma \vdash A : U \rightsquigarrow_d \mathfrak{D}_A \qquad \Gamma, x{:}A \vdash M : B \rightsquigarrow_d \mathfrak{D}_M \\ FV(\lambda^i x{:}A.M) = \overline{x}{:}\overline{A} \qquad \Gamma \vdash \overline{A} : U \rightsquigarrow \overline{\mathsf{A}} \\ \Gamma, x{:}A \vdash M : B \rightsquigarrow \mathsf{M} \qquad \Gamma \vdash \Pi x{:}A.B : U \rightsquigarrow \Pi\mathsf{x}{:}\mathsf{A}.\mathsf{B}\end{array}}{\Gamma \vdash \lambda^i x{:}A.M : \Pi x{:}A.B \rightsquigarrow_d \mathfrak{D}_A \cup \mathfrak{D}_M, \mathfrak{L}_{\mathsf{i}}(\{\overline{\mathsf{x}}{:}\overline{\mathsf{A}}\}, \mathsf{x}{:}\mathsf{A} \mapsto \mathsf{M} : \mathsf{B})}$$

Other rules simply performs extraction structurally. The function extraction process is designed to have a property: if $\Gamma \vdash M : A$, then the function definitions extracted from M always include the definitions extracted from its type A, which is required to prove type preservation. Note that this property is non-trivial when M is an application. Since dependent types allow

any type-level computation, the result type $B[N/x]$ of some application $M\ N$ could contain a function [7] that is not found in either M, N, or the context! So, rule D-Apply should also include functions extracted from the result type of the application.

$$\textsc{d-Apply}\quad \frac{\Gamma \vdash M : \Pi x{:}A.B \rightsquigarrow_d \mathfrak{D}_1 \quad \Gamma \vdash N : A \rightsquigarrow_d \mathfrak{D}_2 \quad \Gamma \vdash B[N/x] : U \rightsquigarrow_d \mathfrak{D}_3}{\Gamma \vdash M\ N : B[N/x] \rightsquigarrow_d \mathfrak{D}_1 \cup \mathfrak{D}_2 \cup \mathfrak{D}_3}$$

I proved that the transformation is type-preserving and correct – well typed source expressions are translated into well-typed target expressions, and the transformation preserves the reduction sequences.

Theorem 4.3 (Correctness) *For all ground types A and values v of type A,*

$$\cdot \vdash M : A \wedge M \triangleright^* v \Longrightarrow \mathfrak{D}_\Gamma \cup \mathfrak{D}_M \vdash \mathsf{M} \triangleright^* \mathsf{v}' \text{ where } \mathsf{v}' \equiv \mathsf{v}.$$

Theorem 4.4 (Type preservation) *For all well-typed programs M,*

$$\Gamma \vdash M : A \Longrightarrow \mathfrak{D}_\Gamma \cup \mathfrak{D}_M; \Gamma \vdash \mathsf{M} : \mathsf{A},$$

where $(\mathfrak{D}_\Gamma, \mathfrak{D}_M) = (\llbracket\Gamma\rrbracket_d, \llbracket M\rrbracket_d)$ *and* $(\Gamma, \mathsf{M}, \mathsf{A}) = (\llbracket\Gamma\rrbracket, \llbracket M\rrbracket, \llbracket A\rrbracket)$.

4.2 A dependently typed assembly language

As analyzed in (§3.2), all dependently typed assembly languages in previous work are not suitable for type-preserving compilation. DTAL [48] does not support fully dependent types and Singleton [47] is incompatible with type-preserving transformations. In this section, I present the design of a dependently typed assembly language that learns from the strengths of its predecessors and overcomes their weakness, with the following design principles:

- **TAL-like syntax.** Following the design of TAL (discussed in [36, 35]), the assembly language is structured in code blocks with a main block where the program starts. Each block is a sequence of RISC-style instructions with a type signature that specifies the preconditions before execution, for example, the types of values stored in registers or on the stack.

- **Separation of logic and assembly.** The assembly language uses dependent types to specify the properties of code blocks with logical statements. Following DTAL, the logical language is separated from the assembly language, to eliminate ambiguous situations like "types depending on mutable registers". As in Singleton, the logic is given by a consistent dependent type theory to ensure soundness.

- **Stack machine.** The operational model of the assembly language is a stack machine, like TALx86 [19], the extended and more practical version of TAL. It makes code generation to stack-based low level languages like JVM and Wasm easier, and many hardware mechanisms expect programs that work with stacks.

[7] Section 3.3.2 of the paper gives an example.

Instructions	I	::=	`pop` | `push` w | `clo` n lab | `jmp` lab | `app` w
Instruction sequences	I_s	::=	`ret` | `halt` | I ; I_s
Words	w	::=	`s(`n`)` | *constants*

Figure 5: Excerpt syntax of the assembly code where lab are code labels and n are natural numbers.

- **Compatible with transformations.** As a target language for type-preserving compilations, it should be able to generate from the intermediate representations of previous type-preserving transformations for dependent types. In particular, it should support direct jumps to fit with CPS transformation and tail calls, and be capable of representing the function labels from defunctionalization.

Defunctionalized Calculus of Constructions (DCC), as introduced in (§4.1), is a promising candidate for the logic of a dependently typed assembly – it is a consistent type theory that captures defunctionalized intermediate representations. Fig. 5 shows an excerpt of the syntax of instructions and here is an example code block in this typed assembly based on DCC.

```
com : [A : U0, B : U0, C : U0]
      {g : {x : A, y : B} → C, f : {x : A} → B, x : A} →
      (g{x, f{x}} : C).
1: push s(0);
2: app s(2);
3: app s(3);
4: ret
```

The type of `com` states that it is invoked with three local variables `A`, `B`, and `C` (which are types). Then, it specifies the types of the top three items on the stack (referred to as `g`, `f`, and `x` with `x` at the top), and claims that the instructions will compute `g x (f x)` and place it on the top of the stack before `com` returns. As you can see, the type signature of `com` resembles the syntax of a DCC label $\mathfrak{L}(\{\overline{x}{:}\overline{A}\}, x{:}A \mapsto M : B)$ – the local variables are like the free variables, the stack (seen as a big dependent pair) is like the function argument, and the resulting computation is like the function body. Here, `s(i)` evaluates to the `i`-th value from the top of the stack. Values `f` and `g` are closures (a pair of a code label and a list of assigned values to its local variables) similar to $\mathfrak{L}\{\overline{M}\}$, that can be created with the `clo` instruction and invoked with `app`. Fig. 6 shows how `com` is executed. The stack machine uses call frames to record the local variables, location of the last frame, and the instruction to return to. It supports direct jumps and tail calls with `jmp` that transfers control directly to code blocks which require no local variables.

The type checker for our new assembly language has two tasks: checking if every code block's signature is well-typed, and if their instruction sequence correctly *implements* the computation as its type specified. For `com`, it checks if `g x (f x)` has type `C`, and if the instructions will compute `g x (f x)` and place it on the top of the stack.

Type checking of the signatures uses exactly the same rules as type checking in DCC. To check if an instruction sequence I_s computes a value v of type A, we need the context of all the type signatures (Ψ), the local variables (Δ), and the stack (Σ), with the judgement $\Psi; \Delta; \Sigma \vdash I_s : (v : A)$. The type checker evaluates the sequence abstractly (like what we did in Fig. 6) and checks if the final result is equivalent to the one in specification. For example, to check if pop; I_s computes v, we pop the top item from the stack and check if I_s computes v. Rules for other instructions follow the same spirit.

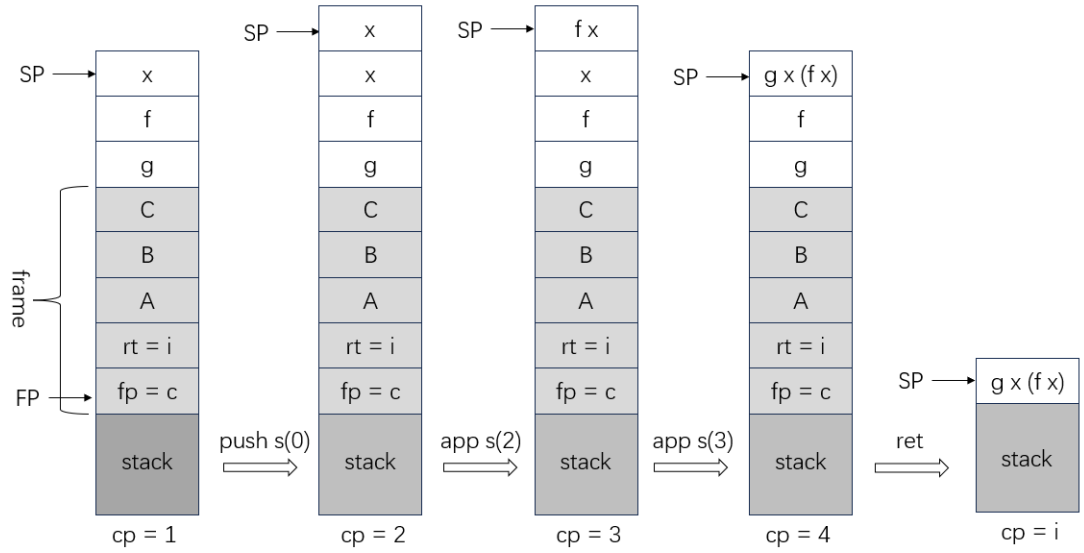


Figure 6: Execution of `com` and components on the stack.

I have prototyped a type checker and an interpreter for this assembly language. The type checker rejects the following code that creates an infinite loop. Although the assembly code implements `(loop[]){}`, the type signature is ill-scoped – a label cannot be mentioned in its own body (see rule DCC-WF-LABEL). I conjecture and plan to prove that the typed assembly language is type safe and well-typed programs always terminate.

```
loop : []{} → ((loop[]){} : U0).
  clo 0 loop
  app s0
  ret
```

4.3 Defunctionalization with quantitative type theory

Defunctionalization with dependent types is achieved with DCC, a dependent type theory specialized for representing the defunctionalized function labels (§4.1), that gives rise to a dependently typed assembly language (§4.2). I believe that this approach can be combined with quantitative types to add precise runtime erasure annotations to the typed assembly. This section outlines the first step, extending defunctionalization to quantitative type theory (QTT).

Defunctionalized Quantitative Type Theory

DCC, the target language of dependently typed defunctionalization, can be extended with quantitative types with a few straightforward modifications. I call the new target language Defunctionalized Quantitative Type Theory (DQTT), with its syntax presented below.

Universes	U	::=	U_i
Expressions	A, B, L, M, N	::=	$x \mid U \mid \Pi x \overset{q}{:} A.B \mid L \mathbin{@} M \mid \mathfrak{L}\{\overline{M}\}$
Type contexts	Γ	::=	$\cdot \mid \Gamma, x \overset{q}{:} A$
Label contexts	$\mathfrak{D}$	::=	$\cdot \mid \mathfrak{D}, \mathfrak{L}(\{\overline{x} \overset{\overline{q}}{:} \overline{A}\}, x \overset{q}{:} A \mapsto M \overset{\sigma}{:} B)$
DCC contexts	$\mathfrak{D};\Gamma$		

As you can see, quantity annotations appear on variables in the context, argument of function types, and inside defunctionalized labels. As in QTT, each variable's associated quantity represents how many times it will be used during execution, where q are elements from a suitable semiring. Each piece of code in the label's body is marked with σ which is either 0 (runtime irrelevant) or 1 (runtime relevant) like term judgements.

Typing rules incorporates quantities in the usual QTT manner (see §3.1), which enforces the zero-needs-nothing property and makes sure that type needs nothing. Most rules look exactly the same as before, as shown below.

DQTT-Var

$$\frac{\vdash \mathfrak{D}; 0\Gamma_1, x \overset{\sigma}{:} A, 0\Gamma_2}{\mathfrak{D}; 0\Gamma_1, x \overset{\sigma}{:} A, 0\Gamma_2 \vdash x \overset{\sigma}{:} A}$$

DQTT-Universe

$$\frac{\vdash \mathfrak{D}; \Gamma}{\mathfrak{D}; \Gamma \vdash U_i \overset{0}{:} U_{i+1}}$$

DQTT-Pi

$$\frac{\mathfrak{D}; 0\Gamma \vdash A \overset{0}{:} U_i \qquad \mathfrak{D}; 0\Gamma, x \overset{q}{:} A \vdash B \overset{0}{:} U_j}{\mathfrak{D}; 0\Gamma \vdash \Pi x \overset{q}{:} A.B \overset{0}{:} U_{\max(i,j)}}$$

The Label rule is more involved, but it is easy to understand as an iterated version of the application rule with arguments $M_1, ..., M_n$. All of M_i are well-typed under some type assignment with possibly different quantities for variables, so, the total usage in the final context is the sum of usages in each context times their designated usages q_i. As for the quantity for codes, it follows the same constraints as in rule DQTT-Apply: if an argument is computationally irrelevant, then either we are constructing an irrelevant expression, or the label will use that variable in an irrelevant way.

DQTT-Apply

$$\frac{\begin{array}{c}\mathfrak{D}; \Gamma_1 \vdash M \overset{\sigma}{:} \Pi x \overset{q}{:} A.B \\ \mathfrak{D}; \Gamma_2 \vdash N \overset{\sigma'}{:} A \\ 0\Gamma_1 = 0\Gamma_2 \\ \sigma' = 0 \Leftrightarrow (q = 0 \vee \sigma = 0)\end{array}}{\mathfrak{D}; \Gamma_1 + q\Gamma_2 \vdash M \mathbin{@} N \overset{\sigma}{:} B[N/x]}$$

DQTT-Label

$$\frac{\begin{array}{c}\mathfrak{L}(\{\overline{x} \overset{\overline{q}}{:} \overline{A}\}, x \overset{q}{:} A \mapsto M \overset{\sigma}{:} B) \in \mathfrak{D} \\ \mathfrak{D}; \Gamma_i \vdash \overline{M} \overset{\overline{q}}{:} \overline{A} \\ \forall i.\ (0\Gamma_1 = 0\Gamma_i) \\ \forall i.\ (q_i = 0 \Leftrightarrow (q = 0 \vee \sigma = 0))\end{array}}{\mathfrak{D}; \Sigma_i\ q_i\Gamma_i \vdash \mathfrak{L}\{\overline{M}\} \overset{\sigma}{:} \Pi x \overset{q}{:} A.B}$$

Defunctionalization transformation

Now we look at the defunctionalization transformation, which includes the term transformation $\Gamma \vdash M \overset{l}{:} A \rightsquigarrow M$ and function extraction $\Gamma \vdash M \overset{l}{:} A \rightsquigarrow_d \mathfrak{D}$. The term transformation has to respect the type-needs-nothing principle when it is finding the free variables required to type the function body in the Lambda rule. In other words, FV finds all the variables required to type the function but the quantity for any free variable that appears only in the types is 0. Other rules are essentially the same as before, such as the rule DQTT-T-Apply shown below.

$$\textsc{dqtt-t-Lambda}\quad \frac{FV(\lambda^i x^q\!:\!A.M) = \overline{x}\ ^{\overline{q}}\!:\ \overline{A} \qquad \Gamma \vdash \overline{x}\ ^{\overline{q}}\!:\ \overline{A} \rightsquigarrow \overline{\mathsf{x}}}{\Gamma \vdash \lambda^i x^q\!:\!A.M\ ^{\sigma}\!:\ \Pi x^q\!:\!A.B \rightsquigarrow \mathfrak{L}_{\mathsf{i}}\{\overline{\mathsf{x}}\}}$$

$$\textsc{dqtt-t-Apply}\quad \frac{\Gamma_1 \vdash M\ ^{\sigma}\!:\ \Pi x^q\!:\!A.B \rightsquigarrow \mathsf{M} \qquad \Gamma_2 \vdash N\ ^{\sigma'}\!:\ A \rightsquigarrow \mathsf{N} \qquad 0\Gamma_1 = 0\Gamma_2 \quad \sigma' = 0 \Leftrightarrow (q = 0 \vee \sigma = 0)}{\Gamma \vdash M\ N\ ^{\sigma}\!:\ B[N/x] \rightsquigarrow \mathsf{M}\ @\ \mathsf{N}}$$

The function extraction follows easily, as long as we remember to get the lambdas from types in the erased mode. For example, $\mathfrak{D}_A$ in rule DQTT-D-LAMBDA is extracted from $0\Gamma \vdash A\ ^{0}\!:\ U$.

$$\textsc{dqtt-d-Lambda}\quad \frac{\begin{array}{c} 0\Gamma \vdash A\ ^{0}\!:\ U \rightsquigarrow_d \mathfrak{D}_A \qquad \Gamma, x^q\!:\!A \vdash M\ ^{\sigma}\!:\ B \rightsquigarrow_d \mathfrak{D}_M \\ FV(\lambda^i x\!:\!A.M) = \overline{x}\ ^{\overline{q}}\!:\ \overline{A} \qquad 0\Gamma \vdash \overline{A}\ ^{0}\!:\ U \rightsquigarrow \overline{\mathsf{A}} \\ \Gamma, x^q\!:\!A \vdash M\ ^{\sigma}\!:\ B \rightsquigarrow \mathsf{M} \qquad 0\Gamma \vdash \Pi x^q\!:\!A.B\ ^{0}\!:\ U \rightsquigarrow \Pi \mathsf{x}^{\mathsf{q}}\!:\!\mathsf{A}.\mathsf{B} \end{array}}{\Gamma \vdash (\lambda^i x^q\!:\!A.M)\ ^{\sigma}\!:\ (\Pi x^q\!:\!A.B) \rightsquigarrow_d \mathfrak{D}_A \cup \mathfrak{D}_M, \mathfrak{L}_{\mathsf{i}}(\{\overline{\mathsf{x}}\ ^{\overline{\mathsf{q}}}\!:\ \overline{\mathsf{A}}\}, \mathsf{x}^{\mathsf{q}}\!:\!\mathsf{A} \mapsto \mathsf{M}^{\sigma}\!:\!\mathsf{B})}$$

Similarly, $\mathfrak{D}_3$ in rule DQTT-D-APPLY is extracted from $0\Gamma \vdash B[N/x]\ ^{0}\!:\ U$. The rules make sure that if a term is in the erased fragment, the function definitions extracted from it are all in the erased fragment as well.

$$\textsc{dqtt-d-Apply}\quad \frac{\begin{array}{c} \Gamma_1 \vdash M\ ^{\sigma}\!:\ \Pi x^q\!:\!A.B \rightsquigarrow_d \mathfrak{D}_1 \\ \Gamma_2 \vdash N\ ^{\sigma'}\!:\ A \rightsquigarrow_d \mathfrak{D}_2 \\ 0\Gamma_1 \vdash B[N/x]\ ^{0}\!:\ U \rightsquigarrow_d \mathfrak{D}_3 \\ 0\Gamma_1 = 0\Gamma_2 \quad \sigma' = 0 \Leftrightarrow (q = 0 \vee \sigma = 0) \end{array}}{\Gamma \vdash M\ N\ ^{\sigma}\!:\ B[N/x] \rightsquigarrow_d \mathfrak{D}_1 \cup \mathfrak{D}_2 \cup \mathfrak{D}_3}$$

For example, the identity function $(\lambda^0 x^0\!:\!U_0.\, \lambda^1 x^1\!:\!A.\, x)$ now defunctionalizes to the following form.

$$\begin{array}{rcl} \mathfrak{D} & ::= & \mathfrak{L}_1(\{\mathsf{A}\ ^{0}\!:\ \mathsf{U}_0\}, \mathsf{x}^1\!:\!\mathsf{A} \mapsto \mathsf{x}^1\!:\!\mathsf{A}), \mathfrak{L}_0(\{\}, \mathsf{A}^0\!:\!\mathsf{U}_0 \mapsto \mathfrak{L}_1\{\mathsf{A}\}^1\!:\!\mathsf{A}\to\mathsf{A}) \\ \mathsf{identity} & ::= & \mathfrak{L}_0\{\} \end{array}$$

Presumably, DQTT would be type-safe and consistent, and the extended defunctionalization transformation would be type-preserving and correct. The proofs are planned in my future work and similar proof techniques as those used for DCC are likely to suffice. For type-safety and consistency, I could follow the method of Boulier et al. [8] to define a type-preserving backward transformation that embeds DQTT into QTT so that the consistency of QTT implies that of DQTT. Type-preservation of the transformation could be established through a variant of the source language with explicit substitutions (see Section 3.4 in [28]).

4.4 Formalization of hereditary substitution

Normalization is an important subject in the study of λ-calculus as it directly relates to the decidability of $\beta\eta$-equality, our ability to test if two terms are equal. In this section, I present an Agda formalization of *hereditary substitution*, a normalization algorithm for simply typed

lambda calculus [8]. Its generalization in contextual modal type theory is related to the notion of my defunctionalized labels, and the study of normalization its into my broader study of design and implementation of dependently typed programming languages.

Syntax and normal forms

We start with the encoding of STLC in Agda: types include a base type ι and arrows types, and contexts are lists of types. Variables are written in de-Bruijn indices, but I will use named variables in the examples to explain better. Terms are indexed over a context and a type, with constructors for variables, lambda abstractions, and applications. Intuitively, $t : \mathsf{Tm}\ \Gamma\ A$ means that $\Gamma \vdash t : A$.

```
data Ty : Set where
  ι : Ty
  _⇒_ : Ty → Ty → Ty

data Con : Set where
  · : Con
  _,_ : Con → Ty → Con

data Var : Con → Ty → Set where
  vz : ∀{Γ α} → Var (Γ , α) α
  vs : ∀{Γ α β} → Var Γ α → Var (Γ , β) α

data Tm : Con → Ty → Set where
  var : ∀{Γ α} → Var Γ α → Tm Γ α
  lam : ∀{Γ α β} → Tm (Γ , α) β → Tm Γ (α ⇒ β)
  app : ∀{Γ α β} → Tm Γ (α ⇒ β) → Tm Γ α → Tm Γ β
```

The η-long-β-short normal forms (Nf) are mutually defined with neutral terms (Ne) that stand for stuck computations – applications of a variable to a list of normals, which is called a spine (Sp). Normal forms contain neutrals at the base type and functions whose body is normal. The syntax of neutrals and normals specifies that the normal forms are fully η-expanded and β-reduced, so, normal forms of functions must begin with a lambda.

```
mutual
  data Nf : Con → Ty → Set where
    lam : ∀{Γ α β} → Nf (Γ , α) β → Nf Γ (α ⇒ β)
    neu : ∀{Γ} → Ne Γ ι → Nf Γ ι

  data Ne : Con → Ty → Set where
    _,_ : ∀{Γ α β} → Var Γ α → Sp Γ α β → Ne Γ β

  data Sp : Con → Ty → Ty → Set where
    · : ∀{Γ α} → Sp Γ α α
    _,_ : ∀{Γ α β γ} → Nf Γ α → Sp Γ β γ → Sp Γ (α ⇒ β) γ
```

A renaming (Ren) is a type-preserving map of variables from a context to another, which applies to terms and normal forms by simple inductions. Common renamings include the

[8] The source code can be found at https://github.com/H-Yulong/HereditarySubstitution/tree/main.

identity, weakening (which is just vs), and swapping that exchanges the order of the two outermost variables. A renaming $\rho : \Gamma \to \Delta$ can be extended to one of type $(\Gamma, \alpha) \to (\Delta, \alpha)$, which maps the extended variable of type α to itself and others according to ρ.

```
Ren : Con → Con → Set
Ren Γ Δ = ∀{α} → Var Γ α → Var Δ α

id : ∀{Γ} → Ren Γ Γ
wk : ∀{Γ α} → Ren Γ (Γ , α)
sw : ∀{Γ β1 β2} → Ren (Γ , β1 , β2) (Γ , β2 , β1)


ext : ∀{Γ Δ α} → Ren Γ Δ → Ren (Γ , α) (Δ , α)
ext ρ vz = vz
ext ρ (vs x) = vs (ρ x)
```

Normalization and hereditary substitution

We can now attempt to define a normalization function by pattern matching on the constructors of Tm. If we see a variable, we apply the η-expansion. In the case of functions, we push the normalization inside the lambda abstraction. If we find an application $M\ N$, we know that the normal form of M must be $\lambda x.M'$ for some normal form M', and the application reduces to the substitution $M'[N'/x]$ where N' is the normal form of N.

However, the capture-avoiding substitution does not preserve normal forms! For example, $(x\ N)[\lambda z.M/x]$ gives $(\lambda z.M)\ N$, which contains a β-redex. So, instead of the conventional substitution, we need to define a hereditary substitution that keeps reducing these β-redices until there is none.

It is easier to define hereditary substitution in the single-variable style, i.e. $M[N]$ that substitutes N for the most recent variable in M. We first see how it works on terms.

```
{-# TERMINATING #-}
_[_]Tm : ∀{Γ α β} → Tm (Γ , β) α → Tm Γ β → Tm Γ α
var vz [ N ]Tm = N
var (vs x) [ N ]Tm = var x
lam M [ N ]Tm = lam (ren M sw [ ren N wk ]Tm)
app M M' [ N ]Tm = app (M [ N ]Tm) (M' [ N ]Tm)
```

We give the definition via pattern matching and suppose the outermost variable is z. In the variable case, if the variable is z then we replace it with N, otherwise we leave it untouched. In the application case, we substitute for both terms. For the case of $\lambda x.M$, we need to apply a swapping renaming to M as the most recent variable is the bound variable x instead of z, and then perform the substitution (note that N is weakened as we moved inside one layer of lambda abstraction). However, this case fails Agda's termination check, which only accepts functions that recurse on syntactically smaller arguments. Agda complains that (ren M sw) is a problematic argument to the recursive call, as it cannot recognize that renaming of a term does not make it grow bigger than before!

There is an elegant solution to this problem. We could delay the renamings on function bodies in a "continuation" and perform them all at once when we hit the variable case. Now, the code becomes:

```
_[_,_]Tm : ∀{Γ Δ α β} → Tm Γ α → Ren Γ (Δ , β) → Tm Δ β → Tm Δ α
var x [ ρ , N ]Tm with ρ x
... | vz = N
... | vs y = var y
lam M [ ρ , N ]Tm = lam (M [ sw ∘ ext ρ , ren N wk ]Tm)
app M M' [ ρ , N ]Tm = app (M [ ρ , N ]Tm) (M' [ ρ , N ]Tm)
```

It takes one more argument ρ as the rest of renamings to do. Intuitively, $M[\rho, N]$ will rename M with ρ, and then perform the single-variable substitution with N. When ρ is the identity renaming, we recover the usual single-variable substitution. In the variable case, we check if $\rho\ x$ gives the most recent variable and perform substitution accordingly. We still substitute for both terms in the application case. In the lambda case, we push the swapping into the continuation by composing it with the extended version of ρ (since we are moving inside one layer of lambda binding). Agda now finds it structurally recursive and admits that it is terminating, as the first argument is decreasing syntactically.

This technique extends to the hereditary substitution scenario, giving a terminating single-variable substitution that preserves normal forms.

```
_[_,_] : ∀{Γ Δ α β} → Nf Γ α → Ren Γ (Δ , β) → Nf Δ β → Nf Δ α
lam t [ ρ , u ] = lam (t [ sw ∘ ext ρ , renNf u vs ])
neu (x , sp) [ ρ , u ] with ρ x
... | vz = u $ (sp < ρ , u >)
... | vs x = neu (x , (sp < ρ , u >))

_<_,_> : ∀{Γ Δ α β γ} → Sp Γ α γ → Ren Γ (Δ , β) → Nf Δ β → Sp Δ α γ
· < ρ , u > = ·
(t , sp) < ρ , u > = (t [ ρ , u ]) , (sp < ρ , u >)

_$_ : ∀{Γ α β} → Nf Γ α → Sp Γ α β → Nf Γ β
t $ · = t
lam t $ (u , sp) = (t [ id , u ]) $ sp
```

The lambda case is the same as before. Recall that a neutral term is a stucked application of a variable to a list of normals. If the variable is substituted by a normal form, the application can carry on iteratively with $. Otherwise, we just map the substitution to the list of normals. Now, we can easily define normalization.

```
nf : ∀{Γ σ} → Tm Γ σ → Nf Γ σ
nf (var x) = nvar x
nf (lam t) = lam (nf t)
nf (app t u) with nf t | nf u
... | lam t' | u' = t' [ id , u' ]
```

Correctness of normalization

I proved that the normalization algorithm is sound and complete, in the sense that:

- Soundness: $\beta\eta$-equivalent terms have equal normal forms.
- Completeness: terms are $\beta\eta$-equivalent to their normal forms.

As a corollary, normalization is idempotent on terms. In fact, normalization is idempotent on all normal forms, and I showed that there exists a unique normal form for every class of $\beta\eta$-equivalent terms. That is, if two normal forms are equivalent, then they are syntactically equal.

Related work

Watkins et al. first introduced hereditary substitution for a normalizer in Concurrent Logical Framework [46]. The formalization presented is improved from the formalization by Keller and Altenkirch [29], which defined a "removal from context" operation to establish termination, making the correctness properties harder to prove.

The alternative normalization algorithm is NbE (discussed in §2.2), which is more efficient in practice but conceptually harder to understand and requires more work to establish correctness. Often, complicated logical relations are required, but all we need is induction for hereditary substitutions.

Future work

The continuation technique extends to contextual modal type theory (CMTT) with meta-variables and meta-substitutions to express code that abstract over contexts [37]. The notion and treatment of meta-variables are similar to the defunctionalized labels in DCC (see §4.1), so, investigating the normalization of CMTT might bring insight into my project for the thesis.

The completeness theorem also produces a chain of rewrites that turns a term into its normal form. The rewrites are subjected to renamings, substitutions, and they have normal forms as well. The formalization of hereditary substitution is a suitable starting point for studying the equational theory of rewrites, providing a mechanized proof that all rewrites between two terms are equal, which is a known theorem from [26, 24].

5 Thesis proposal

In my initial progress, I have developed defunctionalization to remove higher order functions and make function closures explicit with a specialized calculus (§4.1). Then, I showed that such defunctionalized calculus can be used as the reasoning logic of a dependently typed assembly language (§4.2). The next step is extending this process with quantitative types, and I have developed defunctionalization with QTT in (§4.3) as a start.

My final step is deriving QTAL, a quantitatively and dependently typed assembly language, from the intermediate language used in QTT's defunctionalization. The development for QTAL consists of three parts: syntax and semantics, meta-theory, and implementation, each marks a major milestone toward my final goal of providing reliable compilation to dependently typed languages.

Syntax and semantics. Following the design principles outlaid in §4.2, the syntax of QTAL will have two parts: a type theory as its reasoning logic, and a RISC-style assembly instruction set for a stack machine.

The logic will be the Defunctionalized Quantitative Type Theory (DQTT) introduced in §4.3, as it supports both quantitative types and defunctionalized intermediate representations. I will formally define its syntax, substitution, reduction, equality, type judgements, and the defunctionalization transformation into DQTT. It will be extended with datatypes like dependent

pairs, natural numbers, and identity types to demonstrate its ability to deal with various data structures. An extension to inductive families is conceptually simple yet technically complicated (as discussed in §2.1), so, I will not include them into the initial plan.

For the assembly language part, I will define its runtime values, the instruction set, operational semantics, type checking rules which checks if a sequence of instructions implements the computation given by its type signature, and a process that generate assembly code from DQTT expressions. The instructions include a direct `jmp` operation to support tail calls and CPS-based compilation.

A potential risk is that QTT does not have a resource-aware operational semantics, and it is difficult to reason about a program's usage of resources at runtime. I plan to adapt the heap semantics for graded type theory [14] (inspired by [45]) to QTT and then QTAL. If that is not successful, I will use the conventional operational semantics instead.

Meta-theory. I will establish the following meta-theoretical properties for QTAL and the transformations involved.

- **Type safety and consistency.** DQTT, the reasoning logic of DTAL, must be type-safe and consistent as a dependent type theory. QTAL's assembly part should also be type-safe, in the sense that well-typed programs always terminate and will not stuck before it reaches a `halt` instruction or returns a value.
- **Type preservation and correctness.** The two transformations – extended defunctionalization for quantitative types and the assembly code generation – should preserve the input program's type and operational behavior, like Theorem 4.4 for DCC.
- **Erasure.** QTAL should admit type erasure and runtime erasure: removing the type signatures and the 0-usage fragment from a code block should not affect its computation.

These theorems are likely to be proved by inductions, as similar results are obtained by this way for DCC and graded type theory (see Theorem 3.18 in [28]; Theorem 7.5 in [14]). In the case that plain induction does not suffice, I will investigate more powerful proof techniques such as logical relations. I do not plan to mechanize the proofs as it is unlikely to finish within two years – there is no known formalization of QTT for reference and the formalization of graded modal dependent type theory took around 49000 lines of Agda [2]. Instead, the project focuses more on the design and implementation of QTAL.

Implementation. Fig. 7 shows the sketch of a type-preserving compiler for dependently typed languages targeting QTAL. Compared to Fig. 3, it is typed all the way down to the assembly level, and the compiler outputs can be type checked after linking and optimization for correctness. Types and runtime irrelevant components are only removed before generating binary code for execution (after linking and checking is completed). Quantity annotations are inferred at the elaboration stage for source languages without quantitative types.

My implementation starts with the core language QTT, followed by defunctionalization into DQTT, and then assembly code generation into QTAL, followed by optimization stages such as remove needless closure creations. Currently, dependently typed CPS transformation [11] and ANF transformation [30] that exposes control flows suffer from extensionality issues, and type-checking of their generated intermediate representations could be undecidable. So, incorporating CPS-related compilation is left for future work and I will only use defunctionalization in this project. The type checking will use a bidirectional algorithm with

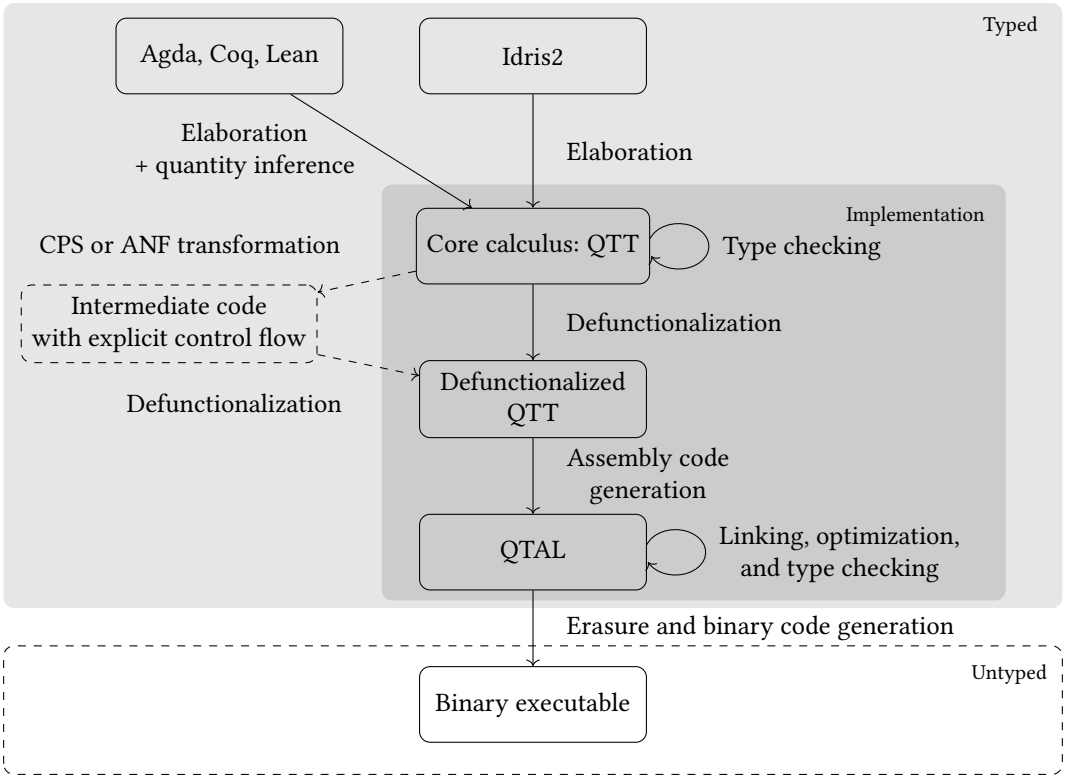


Figure 7: Sketch of type-preserving compiler for dependent types.

normalization by evaluation (see §2.2) for efficiency. Bidirectional checking of QTT is similar to the type checking of Granule [40]. Although NbE for quantitative types is not yet known, it is not difficult to derive following [1, 25]. I do not plan to show the correctness of NbE for QTT, but this topic is worth pursuing if time permits.

I will evaluate the compiler with programs that has implementations in both dependent and non-dependent type systems, for example: sorting, list processing, and array manipulation. The generated QTAL code is compared against code compiled naively without runtime erasure, hand-written fine-tuned algorithms in QTAL, and code generated from non-dependently typed programs.

Table 1 shows my timeplan for the next two years. I aim to submit two acedemic papers, one on the quantitatively and dependently typed assembly language QTAL, another on the type-preserving compilation from QTT to QTAL. My goal is to show that the QTAL code generated from type-preserving compilation can be type-checked for correctness and runtime irrelevant components can be safely erased for efficiency. For the first time, it will be possible to build high-assurance and high-performance compilers for languages with extremely sophisticated type systems.

Time	Scheduled work
2023 Nov - Dec	Define the syntax of DQTT with dependent pairs, natural numbers, and identity types. Work out bidirectional typing and NbE for QTT and DQTT. Develop a heap semantics for QTT. Define the syntax of QTAL (based on DQTT). This includes: the assembly's syntax (instructions, code blocks, and machine word values), type checking rules, and a resource-aware operational semantics.
2024 Jan - Mar	Prove consistency for DQTT and type safety for QTAL's assembly part. Show that QTAL admits both type erasure and runtime erasure. Start implementing the type-checker and interpreter for QTAL.
2024 Apr - Jun	Finish the type checker and interpreter for QTAL. Start writing the paper on QTAL aiming POPL 2025 (which dues in July).
Milestone	*The syntax, operational semantics, and meta-theory of QTAL are fully developed.*
2024 Jul - Sep	Finish the POPL paper on QTAL and submit. Define the defunctionalization transformation for QTT into DQTT, and start proving its correctness and type-preservation. Implement the defunctionalization transformation. Summer break.
Milestone	*My work on QTAL is submitted successfully to POPL.*
2024 Oct - Dec	Define the code generation algorithm from DQTT to QTAL. Finish the correctness and type-preserving proofs for defunctionalization. Implement the code generation and test the whole transformation from QTT to QTAL.
Milestone	*Having the first prototype of a type-preserving compiler targeting QTAL.*
2025 Jan - Mar	Develop tools for separate compilation from QTT to QTAL and test the compiler implementation. Show that the code generation is type-preserving and correct. Buffer time for any unfinished proofs and unexpected situations. *Milestone: All meta-theory involved in QTAL are established.*
2025 Apr - Jun	Add optimization stages to improve the performance of generated QTAL code. Implement a naive compiler without runtime erasure. Write dependently and non-dependently typed programs for evaluations. Compare the efficiency of the generated QTAL code against outputs from the naive compiler, non-dependently typed implementations, and fine-tuned hand-written QTAL programs.
Milestone	*Having imperial evidence that the QTAL compiler produces more efficient code than the compilation without erasure.*
2025 Jul - Sep	Finalizing the implementation, experiment with more optimization stages to further improve. Start writing the paper about compilation from QTT to QTAL at PLDI 2026 (which dues in November). Summer break.
2025 Oct - Dec	Keep preparing for the PLDI paper.
Milestone	*My work on compilation from QTT to QTAL is submitted successfully to PLDI.*

Table 1: Timeplan for the next two years.

References

[1] Andreas Abel. Normalization by evaluation: Dependent types and impredicativity. *Habilitation. Ludwig-Maximilians-Universität München*, 2013.

[2] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. A graded modal dependent type theory with a universe and erasure, formalized. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023.

[3] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science: 6th International Conference, CTCS'95 Cambridge, United Kingdom, August 7–11, 1995 Proceedings 6*, pages 182–199. Springer, 1995.

[4] Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018.

[5] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[6] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, pages 25–37. ACM, 1997.

[7] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 203–211. IEEE Computer Society, 1991.

[8] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 182–194. ACM, 2017.

[9] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda - A functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

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

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

[12] Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[13] Edwin C Brady. *Practical implementation of a dependently typed functional programming language*. PhD thesis, Durham University, 2005.

[14] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021.

[15] The Coq Development Team. The Coq reference manual. https://coq.inria.fr/distrib/current/refman/, October 2022.

[16] Thierry Coquand. 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*, pages 227–236. IEEE Computer Society, 1986.

[17] Thierry Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996.

[18] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[19] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. Talx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, pages 25–35, 1999.

[20] Olivier Danvy. Type-directed partial evaluation. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *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*, pages 242–257. ACM Press, 1996.

[21] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

[22] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5):98:1–98:38, 2022.

[23] Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 26–37, 2002.

[24] Marcelo Fiore and Philip Saville. Coherence and normalisation-by-evaluation for bicategorical cartesian closed structure. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 425–439, 2020.

[25] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *Proc. ACM Program. Lang.*, 3(ICFP):107:1–107:29, 2019.

[26] Barney P. Hilken. Towards a proof theory of rewriting: The simply typed 2lambda-calculus. *Theor. Comput. Sci.*, 170(1-2):407–444, 1996.

[27] Martin Hofmann and Martin Hofmann. Syntax and semantics of dependent types. *Extensional Constructs in Intensional Type Theory*, pages 13–54, 1997.

[28] Yulong Huang and Jeremy Yallop. Defunctionalization with dependent types, 2023.

[29] Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, pages 3–10. ACM, 2010.

[30] Paulette Koronkevich, Ramon Rakow, Amal Ahmed, and William J. Bowman. Anf preserves dependent types up to extensional equality. *Journal of Functional Programming*, 32:e12, 2022.

[31] Zhaohui Luo. *An extended calculus of constructions.* PhD thesis, University of Edinburgh, UK, 1990.

[32] Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016.

[33] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *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*, pages 271–283. ACM Press, 1996.

[34] Benjamin Moon, Harley Eades III, and Dominic Orchard. Graded modal dependent type theory. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 462–490. Springer, 2021.

[35] Greg Morrisett. Typed assembly language. *Advanced Topics in Types and Programming Languages*, pages 137–176, 2002.

[36] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[37] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3):23:1–23:49, 2008.

[38] Lasse R Nielsen. A denotational investigation of defunctionalization. *BRICS Report Series*, 7(47), 2000.

[39] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press Oxford, 1990.

[40] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP):110:1–110:30, 2019.

[41] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 89–98. ACM, 2004.

[42] David Tarditi, J. Gregory Morrisett, Perry Cheng, Christopher A. Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In Charles N. Fischer, editor, *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, pages 181–192. ACM, 1996.

[43] Matus Tejiscak. A dependently typed calculus with pattern matching and erasure inference. *Proc. ACM Program. Lang.*, 4(ICFP):91:1–91:29, 2020.

[44] Amin Timany and Matthieu Sozeau. Consistency of the predicative calculus of cumulative inductive constructions (pCuIC). *CoRR*, abs/1710.03912, 2017.

[45] David N Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1-2):231–248, 1999.

[46] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2003.

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

[48] Hongwei Xi and Robert Harper. A dependently typed assembly language. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, pages 169–180. ACM, 2001.