
# Defunctionalization with Dependent Types*

YULONG HUANG, University of Cambridge, UK
JEREMY YALLOP, University of Cambridge, UK

The *defunctionalization* translation that eliminates higher-order functions from programs forms a key part of many compilers. However, defunctionalization for dependently-typed languages has not been formally studied.

We present the first formally-specified defunctionalization translation for a dependently-typed language and establish key metatheoretical properties such as soundness and type preservation. The translation is suitable for incorporation into type-preserving compilers for dependently-typed languages.

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

Additional Key Words and Phrases: compilation, type preservation, type systems, dependent types

**ACM Reference Format:**
Yulong Huang and Jeremy Yallop. 2023. Defunctionalization with Dependent Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 127 (June 2023), 23 pages. https://doi.org/10.1145/3591241

## 1 INTRODUCTION

Types are increasingly used not merely for *classification* (i.e. in identifying a subset of programs with desirable properties), but for *compilation*. A *type-preserving compiler*, organised as a series of translations between two or more typed languages [e.g. Bowman and Ahmed 2018; Bowman et al. 2018; Morrisett et al. 1999; Tarditi et al. 1996; Xi and Harper 2001], can support features such as type-driven elaboration of source programs into more explicit core calculi [Kovács 2020; Wadler and Blott 1989], translation of disparate source features into a simple uniform core [Sulzmann et al. 2007] and type-driven optimizations [Tarditi et al. 1996]. More generally, type-preserving translations between intermediate languages can increase confidence in correctness of the compilation process [Patrignani et al. 2019].

As type systems increase in sophistication, defining type-preserving presents new challenges. Some of the most significant arise in the compilation of *dependently-typed* languages such as Agda [Bove et al. 2009], Idris [Brady 2013], and Coq [Coq Development Team 2022], whose type systems are sufficiently expressive to support arbitrary computation. It has proved difficult to adapt long-studied translations such as *continuation-passing style* conversion, *closure conversion*, and conversion into *administrative normal form* to the dependently-typed setting [Barthe et al. 1999; Barthe and Uustalu 2002; Bowman and Ahmed 2018; Bowman et al. 2018; Koronkevich et al. 2022].

Another such translation, *defunctionalization*, which eliminates higher-order functions from programs, forms a key part of compilers for several higher-order languages [e.g. Braßel 2011; Pettyjohn et al. 2005; Podlovics et al. 2021; Weeks 2006]. Type-preserving variants of defunctionalization

*We use different colours and fonts to distinguish different languages, and recommend reading this paper in colour.

Authors' addresses: Yulong Huang, University of Cambridge, Cambridge, UK, yh419@cam.ac.uk; Jeremy Yallop, University of Cambridge, Cambridge, UK, jeremy.yallop@cl.cam.ac.uk.




2475-1421/2023/6-ART127
https://doi.org/10.1145/3591241

are available for a variety of type systems [Bell et al. 1997; Nielsen 2000; Pottier and Gauthier 2004]. Defunctionalization is also useful in the compilation of dependently-typed languages, such as Idris [1]. However, to date no type-preserving variant of the defunctionalization translation for dependently-typed languages has been developed.

This work meets that need, introducing a typed defunctionalization translation for a dependently-typed language, and establishing its fundamental properties. As with previous work that has adapted similar program translations to support dependent types, we have encountered and resolved various difficulties that do not arise in simply-typed settings. In particular, the need to preserve universe sizes (used by dependently-typed languages to avoid inconsistencies), and to preserve reduction (used to establish type equality) make a straightforward adaption of the standard defunctionalization unfeasible.

*Contributions.* The central contribution of this paper is the first type-preserving defunctionalization translation for a dependently typed language. In more detail,

- §2 shows that the type-preserving defunctionalization translations used for simply typed languages (§2.1) do not extend to a dependently-typed setting (§2.3), and presents an abstract translation suited to dependently-typed langauges (§2.4).
- §3 has the technical development of our type-preserving defunctionalization translation. §3.3 formally defines the abstract translation and §3.4 and §3.5 establish key meta-theoretical properties such as soundness, type preservation, and consistency.

Finally, §4 summarises related work on type-preserving compilation and on defunctionalization. We also provide an implementation of our translation as supplementary material.

## 2 OVERVIEW

### 2.1 Defunctionalization

The *defunctionalization* translation turns higher-order programs into first-order programs, by replacing the function arrow - → - with a first-order data type - ⇝ -. Defunctionalization replaces each abstraction $\lambda x.e_i$ in the source program with a constructor application $C_i\ \overline{y}$ where $C_i$ is a constructor of - ⇝ - 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 $e_i$.

Here is an example. The polymorphic `compose` function contains three abstractions, here labeled F1, F2, and F3.

```
compose :: (b → c) → (a → b) → (a → c)
compose = λf → λg → λx → f (g x)
                  F3
          F2
F1
```

Defunctionalizing `compose` produces a data type ⇝ with one constructor for each abstraction. Here → separates constructor arguments: `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)
```

Following Pottier and Gauthier [2004], the data type ⇝ produced by defunctionalization is a *generalized algebraic data type* (GADT), in which the return type of each constructor can have a

[1] https://github.com/idris-lang/Idris-dev/blob/v1.3.4/src/IRTS/Defunctionalise.hs

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.

Defunctionalization also produces an operator `$` that maps the constructors of $\rightsquigarrow$ 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 $\rightarrow$ with $\rightsquigarrow$ and F1 with `F1` in `compose` itself:

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

Previous work on defunctionalization in typed settings has examined a variety of languages, from simply-typed [Nielsen 2000] and monomorphizable [Bell et al. 1997] to fully polymorphic [Pottier and Gauthier 2004]. In each case, defunctionalization represents functions using inductive data types, from simple algebraic datatypes for simply-typed functions, to the more powerful *generalized* algebraic data types for polymorphism.

Might the same approach, extended to yet more powerful data types, support defunctionalization of dependently-typed programs? Polymorphic functions abstract over types and defunctionalize to GADTs indexed by types. By analogy, might dependent functions, which abstract over expressions, defunctionalize to inductive families indexed by expressions?

### 2.2 Inductive Families

We briefly recall inductive families and their associated restrictions. Inductive families [Dybjer 1994] generalize both ordinary inductive data types and generalized algebraic data types by permitting indexing by expressions.

The constructor of each inductive family may return an instantiation of the family instantiated with arbitrary indexes. For example, in the following Agda [Norell 2008] definition the constructors of the inductive family of finite sets, Fin, are indexed by natural numbers $n$:

```
data Fin : ℕ → Set₀ where
  fzero : {n : ℕ} → Fin (suc n)
  fsuc : {n : ℕ} → Fin n → Fin (suc n)
```

The constructor fzero constructs an element of type Fin (suc $n$) for any $n$, while fsuc constructs an element of type Fin (suc $n$) from an element of type Fin $n$.

In general, inductive families (without parameters) have the following form:

```
data D : (y₁ : T₁) → … → (yₙ : Tₙ) → Set_d where
    c₁ : A₁
    c₁ : A₂
    …
```

where D is an inductive family in $\mathsf{Set}_d$ indexed by expressions of type $\mathsf{T}_1, \ldots, \mathsf{T}_n$. For each constructor $c_i$, it takes a number of arguments $(z_i : \mathsf{S}_i)$ and constructs an element of type D $t_1$ … $t_n$, where each $t_i$ is an expression of type $\mathsf{T}_i$. Concretely, each $\mathsf{A}_i$ takes the following form:

```
(z₁ : S₁) → … → (zₘ : Sₘ) → D t₁ … tₙ.
```

The arguments to D and $c$ are dependently typed: $T_{i+1}$ can mention $y_1 \ \ldots \ y_i$, and $S_{i+1}$ can mention $z_1 \ \ldots \ z_i$.

To ensure that inductive family definitions are consistent, Agda imposes additional restrictions.

First, *universe checking* rejects inductive definitions with impredicative constructors — that is, definitions whose constructors inhabit a larger universe than the data types themselves. More concretely, for a data type such as D above, the universe of every argument type $S_i$ (i.e. the type of $S_i$) should be smaller than $\mathsf{Set}_d$ to pass the universe check. Without this restriction, inductive families can be used to encode Girard's paradox.

Second, *positivity checking* rejects inductive families that contain references to themselves in non-strictly-positive positions. Without this restriction, inductive families such as the following Fix can be used to build recursive definitions that violate consistency, such as bad:

```
data Fix : Set → Set where
  fix : ∀ {a} → (Fix a → a) → Fix a

bad : ∀ {a} → a
bad = f (fix f) where f : ∀ {a} → Fix a → a
                      f (fix g) = g (fix g)
```

Strict positivity imposes two conditions on the constructor types $A_i$ of an inductive family definition D. First, where D appears, it must not be indexed by expressions involving D itself. Second, in the argument types $S_i$, D must not occur to the left of function arrows.

These requirements around strict positivity and universes are shared by many dependently-typed languages that support inductive families, like Coq's Gallina [Coq Development Team 2022], Lean [de Moura et al. 2015], and Timany and Sozeau's pCuIC [Timany and Sozeau 2017].

### 2.3 Problems Extending Defunctionalization to Dependent Types

At first glance, extending defunctionalization to support dependent functions, targeting inductive families, appears straightforward. As an example, we consider the defunctionalization of the following fully-dependent compose function, written in Agda, with all arguments explicit for clarity:

```
compose : (A : Set) → (B : A → Set) → (C : (x : A) → B x → Set) →
          (f : (y : A) → (z : B y) → C y z) → (g : (x : A) → B x) → (x : A) →
          C x (g x)
compose = λ A → λ B → λ C → λ f → λ g → λ x → f x (g x)
```

Adapting Pottier and Gauthier's recipe, we start by defining an inductive family Π to represent dependent functions, just as the GADT $\rightsquigarrow$ represents non-dependent functions:

```
data Π : (A : Set) → (A → Set) → Set where
```

Each dependent function type $\Pi x{:}A.fx$ (written `(x : A) → f x` in Agda) in the original program will be defunctionalized to `Π A f`.

Next, we add a constructor to Π for each lambda abstraction in the original program. For example, the `F6` constructor corresponds to the innermost abstraction, with free variables `A`, `B`, `C`, `f` and `g`:

```
F6 : (A : Set) →
     (B : Π A (λ _ → Set)) →
     (C : Π A (λ x → Π (B $ x) (λ _ → Set))) →
     (f : Π A (λ y → Π (B $ y) (λ z → C $ y $ z))) →
     (g : Π A (λ x → B $ x)) →
```

```
  Π A (λ x →
  C $ x $ (g $ x))
```

Finally, we add a case for each constructor to the definition of $:

```
F6 A B C f g $ x = f $ x $ (g $ x)
```

Appendix A in the extended version of the paper gives the full definitions. Unfortunately, although these definitions are type-correct, they do not satisfy Agda's additional checks.

First, *universe checking* rejects the definition of `F6` because in the type of $B$ the second argument of Π (i.e. $\lambda\_ \to$ Set) inhabits the universe $\text{Set}_1$, which is larger than Set.

Second, *positivity checking* rejects the definition of `F6` because in the type of $C$, Π is indexed by an expression involving Π.

Finally, Agda's *termination checking* rejects the definition of $ because the case for F6 is not structurally terminating.

*2.3.1 A Simpler Example.* The example above suggests that although defunctionalization apparently extends naturally to dependent types, the extension suffers from consistency problems. In fact, the situation is more grave: even if we do not make use of dependency, the same problems with universes and positivity arise.

For example, here is a simply-typed compose function, based on fixed types $A$, $B$, and $C$:

```
compose : (B → C) → (A → B) → (A → C)
compose = λ f → λ g → λ x → f (g x)
```

Defunctionalizing compose produces an inductive family ⇝ and corresponding *apply* function $:

```
data _⇝_ : Set → Set → Set where
  F1 : (B ⇝ C) ⇝ (A ⇝ B) ⇝ (A ⇝ C)
  F2 : (B ⇝ C) → (A ⇝ B) ⇝ (A ⇝ C)
  F3 : (B ⇝ C) → (A ⇝ B) → (A ⇝ C)

_$_ : ∀ {A B} → (A ⇝ B) → A → B
F1 $ f = F2 f
F2 f $ g = F3 f g
F3 f g $ x = f $ (g $ x)
```

Unfortunately, the simple definition ⇝ suffers from the same problems as the more dependent Π. First, universe checking rejects the constructor F1, because the type $B \rightsquigarrow C$ inhabits the universe $\text{Set}_1$, which is larger than Set. Second, in the type of F1, ⇝ is indexed by ⇝ itself, so the definition fails positivity checking. Finally, the F3 case of $ fails termination checking because the arguments to the recursive call are not structurally smaller than the parameters.

*2.3.2 An Expressivity Mismatch.* We might note that the Agda's restrictions are only fairly crude syntactic approximations of semantic properties, that programs that breach them are not necessarily "incorrect". A similar approach has been taken by Ahrens et al. [2018] (for universe checking), and by Weirich and Casinghino [2010] (for all three checks), among others.

However, we do not favour taking off the safety guards in this way for the code generated by defunctionalization. In our view, 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 Π 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

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

Fig. 1. Defunctionalized simply-typed composition

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.

### 2.4 Abstract Defunctionalization

The examples above suggest that the extension of defunctionalization to dependent types is *type-preserving*. It is also possible to show that it is *meaning-preserving*. As Pottier and Gauthier [2006] observe, when defunctionalization produces a single polymorphic apply function, it coincides with the untyped defunctionalization translation. Pottier and Gauthier use this coincidence to prove that the typed translation is meaning-preserving by lifting a proof about the untyped translation. We might similarly lift the proof to the dependently-typed setting to establish the correctness of the extended translation.

Since the extended defunctionalization translation appears to preserve types and meanings, it is disappointing that it falls foul of Agda's various restrictions. How might we build a translation that does not violate these checks?

We choose to follow the direction taken by Minamide et al. [1996] and Bowman and Ahmed [2018] for *abstract closure conversion*, which studies closure conversion for a specialized target language with new constructs for representing closures and closure types. Closure conversion into these constructs captures the essence of the translation, while avoiding the unnecessary restrictions imposed by more concrete settings. Similarly, we will define a target language, the *Defunctionalized Calculus of Constructions* (DCC), in the style of lambda calculus, but with a new construct for defunctionalized *labels* (representing indexes into a *label context*) in place of lambda abstractions.

Fig. 1 shows the result of defunctionalizing the simply-typed `compose` function to DCC [2], which looks and behaves like the conventional defunctionalization presented in §2.1. In our translation into DCC, each abstraction $\lambda x.e_i$ 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 $e_i$ is stored in a separate *label context* $\mathfrak{D}$ indexed by the label identifier, along with its typing information.

In Fig. 1, 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)`.

It is straightforward to add dependent types to this scheme, but some care is needed to define the transformation and show that it has the desired meta-theoretical properties. In particular, as we shall see, the transformation needs to consider the entire derivation tree rather than just the source language expression (§3.3.2), and we need to use a version of the source language with explicit substitutions (§3.4) to make the type-preservation proof go through. These challenges arise only in defunctionalization in a dependently typed setting, which has not been previously studied.

[2] We assume that $A$, $B$, and $C$ are base types here.

## 3 DEFUNCTIONALIZING WITH DEPENDENT TYPES

Having informally introduced the key concepts and motivated our abstract defunctionalization translation, we now turn to the technical details. The next few sections introduce our source language, the calculus of constructions (§3.1), our target language, the *defunctionalized* calculus of constructions (§3.2), and the defunctionalizion translation that links them (§3.3). We then establish the soundness of the translation (§3.4) and prove the consistency of the target language (§3.5). Proofs of the lemmas and theorems in this section can be found in the extended version of this paper [Huang and Yallop 2023a].

### 3.1 Calculus of Constructions

Our source language is a variant of the Calculus of Constructions (CC) [Coquand and Huet 1988], an expressive dependently-typed lambda calculus that serves as a basis for several programming languages and proof assistants. Our main departure from the original presentation of CC is in following the approach taken by Luo [1990], and by many dependently-typed languages such as Agda, Lean, Coq, and F*, by extending CC with a Martin-Löf style hierarchy of universes.

Here is an example CC definition, compose, which represents the fully dependent composition function for functions $f$ and $g$:

$$\begin{aligned}\text{compose} \quad ::= \quad & \lambda A : U_0.\ \ \lambda B : (\Pi x : A. U_0).\ \ \lambda C : (\Pi x : A. \Pi y : B\ x. U_0). \\ & \lambda f : (\Pi y : A. (\Pi z : B\ y. C\ y\ z)).\ \ \lambda g : (\Pi x : A. B\ x). \\ & \lambda x : A.\ \ f\ x\ (g\ x)\end{aligned}$$

The expression component $\lambda A.\lambda B.\lambda C.\lambda f.\lambda g.\lambda x.f\ x\ (g\ x)$ of this definition is unremarkable; all the interest is in the dependencies of types on arguments. In particular, the result type $C\ y\ z$ of $f$ depends on $f$'s arguments $y$ and $z$ and the result type $B\ x$ of $g$ depends on $g$'s argument $x$. (In a practical programming language, both $y$ and the type arguments $A$, $B$ and $C$ would be passed implicitly, but our minimal calculus does not support implicit arguments.)

Fig. 2a shows the syntax of CC. The expressions of CC are variables $x$ (drawn from an infinite set of names), universes $U$, dependent function types $\Pi x{:}A.B$, applications $L\ M$, and abstractions $\lambda x{:}A.M$. A CC context $\Gamma$ is a telescope of variable-expression pairs.

CC has four judgements:

(1) reduction (Fig. 2b)

$$M \triangleright N$$

(2) type membership (Fig. 2c)

$$\Gamma \vdash M : A$$

(3) context formation (Fig. 2d)

$$\vdash \Gamma$$

(4) equivalence (Fig. 2e)

$$\vdash A \equiv B$$

There is a single reduction rule (Fig. 2b), for $\beta$-reduction, $(\lambda x{:}A.N)\ M \triangleright N[M/x]$. We write $L \triangleright^* M$ to mean that $L$ reduces to $M$ in a sequence with zero or more steps.

CC's rules for typing (Fig. 2c) and context formation (Fig. 2d) are defined by mutual induction.

The type of a variable $x$ is $A$ if $x : A$ is present in the well-formed context $\Gamma$ (TY-VAR). The type of a universe $U_i$ is $U_{i+1}$ (TY-UNIVERSE), and the type of $\Pi x{:}A.B$ is the higher universe among universes of $A$ and $B$ (TY-PI). If $M$ has type $B$ in some context $\Gamma$ extended with $x{:}A$, then $\lambda x{:}A.M$ has the dependent function type $\Pi x{:}A.B$ (TY-LAMBDA). Applications have types $B[N/x]$, since the output

$$\begin{array}{lllll} \text{Expressions} & A, B, L, M, N & ::= & x \mid U \mid \Pi x{:}A.B \mid L\ M \mid \lambda x{:}A.M \\ \text{Universes} & U & ::= & U_i \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \end{array}$$

(a) Syntax

$\boxed{M \triangleright N}$ *(Reduction)*

$$\frac{}{(\lambda x{:}A.N)\ M \triangleright N[M/x]}\ \textsc{red-Beta}$$

(b) Reduction

$\boxed{\Gamma \vdash M : A}$ *(Typing)*

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

$$\frac{\Gamma \vdash A : U_i \qquad \Gamma, x : A \vdash B : U_j}{\Gamma \vdash \Pi x{:}A.B : U_{max(i,j)}}\ \textsc{ty-Pi} \qquad \frac{\Gamma \vdash M : \Pi x{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}\ \textsc{ty-Apply}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}\ \textsc{ty-Lambda} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : U \qquad \vdash A \equiv B}{\Gamma \vdash M : B}\ \textsc{ty-Equiv}$$

(c) Typing

$\boxed{\vdash \Gamma}$ *(Well-formedness)*

$$\frac{}{\vdash \cdot}\ \textsc{wf-Empty} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A : U}{\vdash \Gamma, x : A}\ \textsc{wf-Cons}$$

(d) Context formation

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

$$\frac{\begin{array}{l} L \triangleright^* N \\ M \triangleright^* N \end{array}}{\vdash L \equiv M}\ \textsc{eq-Reduce} \qquad \frac{\begin{array}{l} L \triangleright^* \lambda x{:}A.L' \\ M \triangleright^* M' \\ \vdash L' \equiv M'\ x \end{array}}{\vdash L \equiv M}\ \textsc{eq-Eta1} \qquad \frac{\begin{array}{l} L \triangleright^* L' \\ M \triangleright^* \lambda x{:}A.M' \\ \vdash L'\ x \equiv M' \end{array}}{\vdash L \equiv M}\ \textsc{eq-Eta2}$$

(e) Equivalence

Fig. 2. The Calculus of Constructions (CC)

of a function type may depend on the argument $N$ (TY-Apply). Finally, if an expression $M$ has type $A$ and $A$ is equivalent to $B$, then $M$ also has type $B$ (TY-Equiv).

A context $\Gamma$ is *well-formed* (written $\vdash \Gamma$) if every variable in it is associated with a valid type — that is, the associated expression's type is a universe in the context $\Gamma$.

We make use of two shorthands, writing $\Gamma \vdash A : U$ to mean that $\Gamma \vdash A : U_i$ for *some* $i$ (which means that $A$ is a type), and $A \to B$ to stand for the $\Pi$-type $\Pi x{:}A.B$ where $B$ does not depend on $x$. For simplicity, we omit base types such as the unit type 1 and the natural numbers $\mathbb{N}$ from the formal definition, but we will use them freely in examples.

In CC, two expressions are *equivalent* (Fig. 2e) if they reduce to the same expression (EQ-Reduce) or are $\eta$-equivalent as defined by two symmetric rules EQ-Eta1 and EQ-Eta2. Under EQ-Eta1, $L$

and $M$ are equivalent if $L$ reduces to an abstraction $\lambda x{:}A.L'$, $M$ reduces to some $M'$, and $L' \equiv M'\ x$, and Eq-Eta2 corresponds symmetrically [Bowman and Ahmed 2018; Bowman et al. 2018].

One useful property of CC is as follows: if $\Gamma \vdash M : A$, then $\Gamma \vdash A : U$. Furthemore, CC is type safe and consistent, and type-checking in CC is decidable [Coquand and Huet 1988; Luo 1990].

### 3.2 Defunctionalized Calculus of Constructions

Fig. 3a shows the syntax of our target language, the Defunctionalized Calculus of Constructions (DCC). As in CC, DCC expressions include variables $\mathsf{x}$, universes $\mathsf{U}$, dependent function types $\Pi \mathsf{x}{:}\mathsf{A}.\mathsf{B}$, and applications $\mathsf{L}\ @\ \mathsf{M}$. Unlike CC, DCC contains first-class function labels $\mathfrak{L}\{\overline{\mathsf{M}}\}$ instead of lambda abstractions.

A label expression $\mathfrak{L}\{\overline{\mathsf{M}}\}$ is a label name $\mathfrak{L}$ supplied with a list of zero or more expressions $\overline{\mathsf{M}}$ (standing for $\mathsf{M}_1, \cdots, \mathsf{M}_n$) assigned to its free variables. Label names $\mathfrak{L}_1, \mathfrak{L}_2, \cdots$ are disjoint from variable names, as we emphasize using a different font.

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

DCC has four judgements:

(1) reduction (Fig. 3b)

$$\mathfrak{D} \vdash \mathsf{M} \triangleright \mathsf{N}$$

(2) type membership (Fig. 3c)

$$\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A}$$

(3) context formation (Fig. 3d)

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

(4) equivalence (Fig. 3e)

$$\mathfrak{D} \vdash \mathsf{A} \equiv \mathsf{B}$$

*3.2.1 Reduction.* There is a single reduction rule (Fig. 3b), for label application: the application of the label $\mathfrak{L}\{\overline{\mathsf{M}}\}$ to the argument $\mathsf{N}$ reduces to $\mathsf{L}[\overline{\mathsf{M}}/\overline{\mathsf{x}}, \mathsf{N}/\mathsf{x}]$, where $\mathsf{L}$ is the body of the entry for $\mathfrak{L}$ in the label context and $\overline{\mathsf{M}}$ is the closure of $\mathfrak{L}$. A reduction sequence is noted as $\mathfrak{D} \vdash \mathsf{M} \triangleright^* \mathsf{N}$, which means $\mathsf{M}$ reduces to $\mathsf{N}$ in zero or more steps.

Substitutions for variables, universes, Π-types and applications in DCC follow the conventional definition. Substitutions for labels are

$$\mathfrak{L}\{\overline{\mathsf{M}}\}[\mathsf{N}/\mathsf{x}] \triangleq \mathfrak{L}\{\overline{\mathsf{M}}[\mathsf{N}/\mathsf{x}]\},$$

where $\overline{\mathsf{M}}[\mathsf{N}/\mathsf{x}]$ is syntactic sugar for $\mathsf{M}_1[\mathsf{N}/\mathsf{x}], \cdots, \mathsf{M}_n[\mathsf{N}/\mathsf{x}]$.

*3.2.2 Type Judgements.* DCC's type judgements are of the form $\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A}$, and typing rules are given in Fig. 3c. Rules for variables, universes, Π-types, applications, and conversion are identical to their counterpart rules in CC, so we focus on the rule for labels. A label term $\mathfrak{L}\{\overline{\mathsf{M}}\}$ is well-typed in $\mathfrak{D}; \Gamma$ if the following conditions are satisfied.

(1) The context $\mathfrak{D}; \Gamma$ is *well-formed*.
(2) $\mathfrak{L}(\{\overline{\mathsf{x}} : \overline{\mathsf{A}}\}, \mathsf{x}{:}\mathsf{A} \mapsto \mathsf{M} : \mathsf{B})$ is present in $\mathfrak{D}$.
(3) The length of the two lists $\overline{\mathsf{M}}$ and $\overline{\mathsf{x}}{:}\overline{\mathsf{A}}$ are equal.

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

(a) Syntax

$\boxed{\mathfrak{D} \vdash M \triangleright N}$ *(Reduction)*

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

(b) Reduction

$\boxed{\mathfrak{D}; \Gamma \vdash M : A}$ *(Typing)*

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

$$\frac{\mathfrak{D}; \Gamma \vdash A : U_i \quad \mathfrak{D}; \Gamma, x : A \vdash B : U_j}{\mathfrak{D}; \Gamma \vdash \Pi x{:}A.B : U_{\max(i,j)}}\ \textsc{d-ty-Pi} \qquad \frac{\mathfrak{D}; \Gamma \vdash M : \Pi x{:}A.B \quad \mathfrak{D}; \Gamma \vdash N : A}{\mathfrak{D}; \Gamma \vdash M \mathbin{@} N : B[N/x]}\ \textsc{d-ty-Apply}$$

$$\frac{\mathfrak{D}; \Gamma \vdash \overline{M} : \overline{A} \quad \mathfrak{L}(\{\overline{x} : \overline{A}\}, x{:}A \mapsto M : B) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \mathfrak{L}\{\overline{M}\} : \Pi x{:}A[\overline{M}/\overline{x}].B[\overline{M}/\overline{x}]}\ \textsc{d-ty-Label} \qquad \frac{\mathfrak{D}; \Gamma \vdash M : A \quad \mathfrak{D}; \Gamma \vdash B : U_i \quad \mathfrak{D} \vdash A \equiv B}{\mathfrak{D}; \Gamma \vdash M : B}\ \textsc{d-ty-Equiv}$$

(c) Typing

$\boxed{\vdash \mathfrak{D}; \Gamma}$ *(Well-formedness)*

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

(d) Context and label context formation

$\boxed{\mathfrak{D} \vdash M \equiv N}$ *(Equivalence)*

$$\frac{\mathfrak{D} \vdash L \triangleright^* N \quad \mathfrak{D} \vdash M \triangleright^* N}{\mathfrak{D} \vdash L \equiv M}\ \textsc{d-eq-Reduce}$$

$$\frac{\mathfrak{D} \vdash L \triangleright^* \mathfrak{L}\{\overline{N}\} \quad \mathfrak{D} \vdash M \triangleright^* M' \quad \mathfrak{L}(\{\overline{x} : \overline{A}\}, x{:}A \mapsto N : B) \in \mathfrak{D} \quad \mathfrak{D} \vdash N[\overline{N}/\overline{x}] \equiv M' \mathbin{@} x}{\mathfrak{D} \vdash L \equiv M}\ \textsc{d-eq-Eta1}$$

$$\frac{\mathfrak{D} \vdash L \triangleright^* L' \quad \mathfrak{D} \vdash M \triangleright^* \mathfrak{L}\{\overline{N}\} \quad \mathfrak{L}(\{\overline{x} : \overline{A}\}, x{:}A \mapsto N : B) \in \mathfrak{D} \quad \mathfrak{D} \vdash L' \mathbin{@} x \equiv N[\overline{N}/\overline{x}]}{\mathfrak{D} \vdash L \equiv M}\ \textsc{d-eq-Eta2}$$

(e) Equivalence

Fig. 3. The Defunctionalized Calculus of Constructions (DCC)

(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.

The DCC judgement for well-formed contexts is $\vdash \mathfrak{D};\Gamma$ and its rules are given in Fig. 3d. A context is *well-formed* if every variable in the type context is associated with a valid type (in the previous context $\mathfrak{D};\Gamma$), 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 and the free variables in $M$ (namely $\mathfrak{D}; \overline{x} : \overline{A}, x : A$).

Two terms $L$ and $M$ are equivalent (Fig. 3e) if they both reduce to the same term $N$ in a reduction sequence or they are $\eta$-equivalent. DCC's $\eta$-equivalence rules are similar to that of CC. Rule (D-EQ-ETA1) defines that $L$ and $M$ are equivalent if $L$ reduces to a label $\mathfrak{L}\{\overline{N}\}$, $M$ reduces to $M'$, $\mathfrak{L}(\{\overline{x} : \overline{A}\}, x : A \mapsto N : B)$ is found in the label context $\mathfrak{D}$, and $M' \mathbin{@} x$ is equivalent to $N[\overline{N}/\overline{x}]$. Rules (D-EQ-ETA1) and (D-EQ-ETA2) are symmetrical.

Both the type context and the label context have the weakening property: a well-typed expression is still well-typed in an extended type or label context (by induction on the type derivation rules).

Lemma 3.1 (Type weakening). *If* $\mathfrak{D};\Gamma \vdash M : A$, $\mathfrak{D};\Gamma \vdash B : U_i$, *and* $x$ *is fresh, then* $\mathfrak{D};\Gamma, x : B \vdash M : A$.

Lemma 3.2 (Label weakening). *If* $\mathfrak{D};\Gamma \vdash M : C$, $\mathfrak{D}; \overline{x} : \overline{A}, x : A \vdash N : B$, *and* $\mathfrak{L}_i$ *is fresh, then* $\mathfrak{D}, \mathfrak{L}_i(\{\overline{x} : \overline{A}\}, x : A \mapsto N : B); \Gamma \vdash M : C$.

DCC is type-safe and consistent (we establish these properties in §3.5). In addition, it is sufficiently expressive to support the compose function, but we must write it in defunctionalized style, since the calculus does not support lambda abstraction. There is one entry in the label context $\mathfrak{D}$ for each $\lambda$ in the CC definition of compose:

$$\begin{aligned}\mathfrak{D} \quad ::= \quad & \mathfrak{L}_5(\{A, B, C, f, g\}, x : A \mapsto (f \mathbin{@} x) \mathbin{@} g \mathbin{@} x : (C \mathbin{@} x) \mathbin{@} g \mathbin{@} x),\\ & \cdots,\\ & \mathfrak{L}_1(\{A : U_0\}, B : (\Pi x{:}A.U_0) \mapsto \mathfrak{L}_2\{A, B\} : \Pi C.\Pi f.\Pi g.\Pi x.(C \mathbin{@} x) \mathbin{@} g \mathbin{@} x),\\ & \mathfrak{L}_0(\{\}, A : U_0 \mapsto \mathfrak{L}_1\{A\} : \Pi B.\Pi C.\Pi f.\Pi g.\Pi x.(C \mathbin{@} x) \mathbin{@} g \mathbin{@} x)\end{aligned}$$

and the definition of compose itself is simply a projection of a closed label from the context:

$$\mathsf{compose} \quad ::= \quad \mathfrak{L}_0\{\}$$

The full definition appears in Appendix B in the extended version of the paper.

### 3.3 The Defunctionalization Translation

Fig. 4 shows the translation. It consists of two parts: a transformation $[\![-]\!]$ for expressions and a meta-function $[\![-]\!]_d$ that extracts function definitions from the source program. The expression transformation produces the target program and the meta-function $[\![-]\!]_d$ gives a label context.

$\boxed{\Gamma \vdash M : A \rightsquigarrow \mathsf{M}}$ *(Expression transformation)*

$$\frac{}{\Gamma \vdash x : A \rightsquigarrow \mathsf{x}}\ \textsc{t-Var} \qquad \frac{}{\Gamma \vdash U_i : U_{i+1} \rightsquigarrow \mathsf{U_i}}\ \textsc{t-Universe}$$

$$\frac{\Gamma \vdash A : U_i \rightsquigarrow \mathsf{A} \quad \Gamma, x : A \vdash B : U_j \rightsquigarrow \mathsf{B}}{\Gamma \vdash \Pi x{:}A.B : U_{max(i,j)} \rightsquigarrow \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{B}}\ \textsc{t-Pi} \qquad \frac{\Gamma \vdash L : \Pi x{:}A.B \rightsquigarrow \mathsf{L} \quad \Gamma \vdash M : A \rightsquigarrow \mathsf{M}}{\Gamma \vdash L\ M : B[M/x] \rightsquigarrow \mathsf{L}\ @\ \mathsf{M}}\ \textsc{t-Apply}$$

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

(a) Defunctionalization of expressions

$\boxed{\Gamma \vdash M : A \rightsquigarrow_d \mathfrak{D}}$ *(Function extraction)*

$$\frac{\Gamma \vdash A : U \rightsquigarrow_d \mathfrak{D}}{\Gamma \vdash x : A \rightsquigarrow_d \mathfrak{D}}\ \textsc{d-Var} \qquad \frac{}{\Gamma \vdash U_i : U_{i+1} \rightsquigarrow_d \cdot}\ \textsc{d-Universe}$$

$$\frac{\Gamma \vdash A : U_i \rightsquigarrow_d \mathfrak{D}_A \quad \Gamma, x : A \vdash B : U_j \rightsquigarrow_d \mathfrak{D}_B}{\Gamma \vdash \Pi x{:}A.B : U_{max(i,j)} \rightsquigarrow_d \mathfrak{D}_A \cup \mathfrak{D}_B}\ \textsc{d-Pi}$$

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

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

(b) Extraction of function definitions

Fig. 4. The Defunctionalization Translation

### 3.3.1 *Expression Transformation.*

Definition 3.3. *The expression transformation* $[\![-]\!]$ *takes a well-typed term in CC and an implicit argument of that term's type derivation. We define* $[\![M]\!] \triangleq \mathsf{M}$*, where* $\mathsf{M}$ *is given by a new judgement of the form* $\Gamma \vdash M : A \rightsquigarrow \mathsf{M}$ *(Fig. 4a).*

The transformation simply transcribes the variables, universes, Π-types, applications, and base types and values in CC to their counterparts in DCC compositionally. Functions in the source language are translated into labels in the target language.

Defunctionalization requires a unique correspondence between each label and each source-program function. We use a convention that every lambda in the transformation's input $M$ is tagged with a unique identifier $i$ ($i \in \mathbb{N}$), and its corresponding label's name is $\mathfrak{L}_\mathsf{i}$.

The transformation turns a function $\lambda^i x{:}A.M$ into a label $\mathfrak{L}_i\{\overline{x}\}$, where $\overline{x}$ come from the function's free variables $\overline{x}$ (T-Lambda). The meta-function FV (see Definition 3.4) computes all free variables and their types involved in a well-typed CC-expression. Note that FV is different from *fv*, the conventional free variable function that computes all the *unbound variables* in an expression. In dependently typed languages, the type of a free variable may contain other free variables, and their types may still contain other free variables, and so on! Therefore, $\mathrm{FV}(M)$ must recursively work out all the variables needed for $M$ to be well-typed.

Definition 3.4. *$FV(M)$ takes $\Gamma \vdash M : A$, the type judgement of $M$, as an implicit argument. It firstly computes all the unbound variables $x_1, \cdots, x_n$ in $M$ and in $A$, then calls itself recursively on types of these variables, and finally returns the union of all free variables and their types it found.*

$$\begin{array}{rcl} FV(M) & = & FV(A_1) \cup \cdots \cup FV(A_n) \cup \Gamma_{fv} \\ & \textit{where} & \mathrm{fv}\ (M) \cup \mathrm{fv}\ (A) = x_1, \cdots, x_n \\ & & \Gamma \vdash x_1 : A_1, \cdots, \Gamma \vdash x_n : A_n \\ & & \Gamma_{fv} \triangleq x_1 : A_1, \cdots, x_n : A_n. \end{array}$$

Here, the union of two type contexts $\Gamma_1 \cup \Gamma_2$ is $\Gamma_1$ appended with all the variable-expression pairs $x : A$ that only appear in $\Gamma_2$, preserving their order. Intuitively, $FV(M)$ computes all the variables needed to correctly type $M$. Therefore, $M$ is still well-typed in its free-variable context $FV(M)$.

Lemma 3.5. *If $\Gamma \vdash M : A$, then $FV(M) \vdash M : A$.*

### 3.3.2 Extracting Function Definitions.

Definition 3.6. *$[\![-]\!]_d$ takes a well-typed CC term and implicitly its type derivation. We define $[\![M]\!]_d \triangleq \mathfrak{D}$, where $\mathfrak{D}$ is given by a new judgement of the form $\Gamma \vdash M : A \rightsquigarrow_d \mathfrak{D}$ (Fig. 4b).*

In a simply typed system, the only thing $[\![-]\!]_d$ has to do is finding every function $\lambda^i x{:}A.M$ in the source program and placing them in the label context $\mathfrak{D}$ in the following form

$$\mathfrak{L}(\{\overline{\mathsf{x} : \mathsf{A}}\}, \mathsf{x}{:}\mathsf{A} \mapsto \mathsf{M} : \mathsf{B})$$

where $\{\overline{\mathsf{x}{:}\mathsf{A}}\}$, $\mathsf{x} : \mathsf{A}$, $\mathsf{M}$, and $\mathsf{B}$ respectively correspond to the free variables $(\overline{x : A})$ in the function, the bound variable $x : A$, the function body $M$, and the return type $B$.

Alas, types may index over functions in our dependent type theory, and functions may appear in the type of an expression, even if the expression itself does not contain that function! For example, consider the following triple $(\Gamma, M, N)$ in CC (with built-in natural numbers and addition).

$$\begin{array}{rcl} \Gamma & \triangleq & \cdot,\ A : (Nat \to Nat) \to U_0,\ a : \Pi f : (Nat \to Nat).A\ (\lambda n : Nat.1 + (f\ n)) \\ M & \triangleq & a\ (\lambda x : Nat.1 + x) \\ N & \triangleq & A\ (\lambda n : Nat.2 + n) \end{array}$$

$A$ is a family of types indexed by $Nat \to Nat$ functions and $a\ f$ constructs an element of type $A\ (\lambda n : Nat.1 + (f\ n))$. According to the rule (T-Apply), the inferred type of $M$ is

$$\begin{array}{l} (A\ (\lambda n : Nat.1 + (f\ n)))[(\lambda x : Nat.1 + x)/f] \\ = A\ (\lambda n : Nat.(1 + (\lambda x : Nat.1 + x)\ n)), \end{array}$$

which reduces to $A\ (\lambda n : Nat.2 + n)$. We have $\Gamma \vdash M : N$, yet $N$ contains a function that is not in $\Gamma$ or $M$! We should include this new function in $\mathfrak{D}$, as it guarantees that we will never be in a situation where we need a non-existent label in $[\![M]\!]_d$ to type $[\![M]\!]$. In other words, the transformation defunctionalizes not just the source-language expression, but its entire type derivation tree.

Hence, we arrive at the rules in Fig. 4b. Type derivations of universes do not involve functions at all (D-Universe). Function definitions in a variable $x$ are just the definitions in its type $A$ (D-Var).

Definitions in a dependent function type $\Pi x : A.B$ are the *union* of definitions in $A$ and $B$ (D-Pi). The union here is defined in the same way as the union of contexts (see Definition 3.4), and there is no ambiguity since different functions correspond to different label names.

Definitions in an application $M\ N$ are the union of definitions in $M$, $N$, and $B[N/x]$, since the substitution $B[N/x]$ may create new function definitions (D-Apply). For a lambda abstraction $\lambda^i x{:}A.M$, the definitions it contains are the union of definitions in $M$ and in $A$ appended with $\mathfrak{L}_i$, the definition of itself (D-Lambda). If $M$ has type $B$ by the conversion rule, then the definitions involved in the derivation of $\Gamma \vdash M : B$ are the union of definitions in the derivation of $\Gamma \vdash M : A$ and definitions in $B$ (D-Equiv).

We define the *subset relation* of label contexts to help state further definitions and theorems.

Definition 3.7. *For two well-formed label contexts $\mathfrak{D}_1$ and $\mathfrak{D}_2$, $\mathfrak{D}_1 \subseteq \mathfrak{D}_2$ if for all $\mathfrak{L}_i(\{\overline{x} : \overline{A}\}, x{:}A \mapsto N : B)$ in $\mathfrak{D}_1$, $\mathfrak{L}_i(\{\overline{x} : \overline{A}\}, x{:}A \mapsto N : B)$ is also in $\mathfrak{D}_2$.*

The notion of subsets gives a stronger weakening property to DCC: a well-typed expression is still well-typed in a larger label context.

Lemma 3.8 (Label context weakening (subsets)). *If $\mathfrak{D}_1; \Gamma \vdash M : A, \vdash \mathfrak{D}_2$, and $\mathfrak{D}_1 \subseteq \mathfrak{D}_2$, then $\mathfrak{D}_2; \Gamma \vdash M : A$.*

Since the transformation defunctionalizes the entire type derivation tree of an expression, if $\Gamma \vdash M : A$, then all elements in $[\![A]\!]_d$ are also in $[\![M]\!]_d$. We can prove this property by induction on the type derivation rules.

Lemma 3.9. *For any well-typed expression $\Gamma \vdash M : A$ in CC, $[\![A]\!]_d \subseteq [\![M]\!]_d$.*

The expression transformation and the process of extracting function definitions ($[\![-]\!]$ and $[\![-]\!]_d$) act pointwise on CC contexts. In other words,

$$\begin{array}{ll} [\![\cdot]\!] \triangleq \cdot, & [\![\Gamma, x:A]\!] \triangleq [\![\Gamma]\!], x : [\![A]\!], \\ [\![\cdot]\!]_d \triangleq \cdot, & [\![\Gamma, x:A]\!]_d \triangleq [\![\Gamma]\!]_d \cup [\![A]\!]_d. \end{array}$$

Now, we can see that the (tagged) composition function $\lambda^0 A.\lambda^1 B.\lambda^2 C.\lambda^3 f.\lambda^4 g.\lambda^5 x.f\ x\ (g\ x)$ transforms to $\mathfrak{L}_0\{\}$, a label with no free variables supplied, since the function is closed. The label context $\mathfrak{D}$ for composition can be derived from the function extraction judgements with the sketch derivation tree shown below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A,B,C,f,g \vdash \lambda^5 x.f\ x\ (g\ x) \rightsquigarrow_d \mathfrak{L}_5(\{A,B,C,f,g\}, x : \_ \mapsto (f\ @\ x)\ @\ g\ @\ x : \_)}}{A,B,C,f \vdash \lambda^4 g.\lambda^5 x.f\ x\ (g\ x) \rightsquigarrow_d \cdots, \mathfrak{L}_4(\{A,B,C,f\}, g : \_ \mapsto \mathfrak{L}_5\{A,B,C,f,g\} : \_)}\ \text{D-Lambda}}{A,B,C \vdash \lambda^3 f.\lambda^4 g.\lambda^5 x.f\ x\ (g\ x) \rightsquigarrow_d \cdots, \mathfrak{L}_3(\{A,B,C\}, f : \_ \mapsto \mathfrak{L}_4\{A,B,C,f\} : \_)}\ \text{D-Lambda}}{A,B \vdash \lambda^2 C.\lambda^3 f.\lambda^4 g.\lambda^5 x.f\ x\ (g\ x) \rightsquigarrow_d \cdots, \mathfrak{L}_2(\{A,B\}, C : \_ \mapsto \mathfrak{L}_3\{A,B,C\} : \_)}\ \text{D-Lambda}}{A \vdash \lambda^1 B.\lambda^2 C.\lambda^3 f.\lambda^4 g.\lambda^5 x.f\ x\ (g\ x) \rightsquigarrow_d \cdots, \mathfrak{L}_1(\{A\}, B : \_ \mapsto \mathfrak{L}_2\{A,B\} : \_)}\ \text{D-Lambda}}{\cdot \vdash \lambda^0 A.\lambda^1 B.\lambda^2 C.\lambda^3 f.\lambda^4 g.\lambda^5 x.f\ x\ (g\ x) \rightsquigarrow_d \cdots, \mathfrak{L}_0(\{\}, A : \_ \mapsto \mathfrak{L}_1\{A\} : \_)}\ \text{D-Lambda}$$

### 3.4 Soundness

We consider dependently typed defunctionalization *correct* if for all base types $A$, values $v$, and programs $M$ of type $A$,

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

In other words, if a closed program $M$ evaluates to a base-type value $v$, then $M$ evaluates to a base-type value $v'$ that is equivalent to $v$. This property follows as a corollary of the *preservation of*

$$\textit{Expressions} \quad ::= \quad \cdots \mid M\{x \mapsto N\}$$

$\boxed{\Gamma \vdash M : A}$ *(Typing)*

$$\frac{\Gamma \vdash M : \Pi x{:}A.\, B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\{x \mapsto N\}}\ \textsc{s-ty-Apply} \qquad \frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash N : A}{\Gamma \vdash M\{x \mapsto N\} : B\{x \mapsto N\}}\ \textsc{s-ty-Subst}$$

$\boxed{M \triangleright N}$ *(Reduction)*

$$\frac{}{x\{y \mapsto N\} \triangleright x}\ \textsc{s-red-Var1} \qquad \frac{}{x\{x \mapsto N\} \triangleright N}\ \textsc{s-red-Var2}$$

$$\frac{}{U_i\{x \mapsto N\} \triangleright U_i}\ \textsc{s-red-Universe} \qquad \frac{}{(\lambda x{:}A.M)\ N \triangleright M\{x \mapsto N\}}\ \textsc{s-red-Beta}$$

$$\frac{}{(L\ M)\{x \mapsto N\} \triangleright (L\{x \mapsto N\})\ (M\{x \mapsto N\})}\ \textsc{s-red-Apply}$$

$$\frac{}{((\lambda x{:}A.M)\{\overline{x} \mapsto \overline{M}\})\ N \triangleright (M\{\overline{x} \mapsto \overline{M}\})\{x \mapsto N\}}\ \textsc{s-red-Closure}$$

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

$$\frac{\begin{array}{l} L \triangleright^* (\lambda x{:}A.N)\{\overline{y} \mapsto \overline{N}\} \\ M \triangleright^* M' \\ \vdash N\{\overline{y} \mapsto \overline{N}\} \equiv M'\ x \end{array}}{\vdash L \equiv M}\ \textsc{s-eq-Closure1} \qquad \frac{\begin{array}{l} M \triangleright^* (\lambda x{:}A.N)\{\overline{y} \mapsto \overline{N}\} \\ L \triangleright^* L' \\ \vdash L'\ x \equiv N\{\overline{y} \mapsto \overline{N}\} \end{array}}{\vdash L \equiv M}\ \textsc{s-eq-Closure2}$$

Fig. 5. New syntax and rules in $\mathrm{CC}_S$

*reduction sequences*, which states that

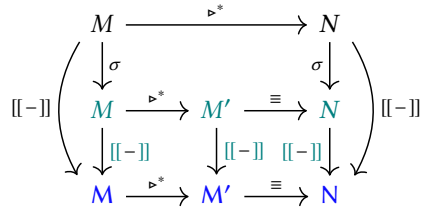
$$M \triangleright^* N \implies \mathfrak{D}_\Gamma \cup \mathfrak{D}_M \cup \mathfrak{D}_N \vdash \mathsf{M} \triangleright^* \mathsf{M}', \quad \text{where } \mathfrak{D}_\Gamma \cup \mathfrak{D}_M \cup \mathfrak{D}_N \vdash \mathsf{M}' \equiv \mathsf{N}.$$

Ideally, we could show this property by showing that the transformation preserves all small-step reductions $M \triangleright N$, followed by an induction on the number of reduction steps in a sequence. However, CC's meta-language substitution $(\lambda x : A.M)[N/y]$ creates a new function definition when it substitutes an expression into a free variable of a function. So, for a reduction sequence $M_1 \triangleright \cdots \triangleright M_n$ in CC, some $M_i$ may contain function definitions that do not exist in $M_1$ or $M_n$. Consequently, not all $M_i$ translate into well-typed DCC expressions $\mathsf{M}_i$ in $(\mathfrak{D}_\Gamma \cup \mathfrak{D}_{M_1} \cup \mathfrak{D}_{M_n})$;$\Gamma$, which makes the standard approach infeasible. Moreover, preservation of reduction sequences is a key lemma for showing type preservation, since CC's typing rules involve equivalence and the equivalence rule (EQ-Reduce) is defined with reductions.

Fortunately, meta-theoretic substitution is the only means of creating new function definitions in CC's reduction sequences. There would be no problem if the source language did not evaluate substitutions into functions but kept them as primitive expressions. To apply this observation we define a helper language $\mathrm{CC}_S$, which is an extension of CC with *explicit substitutions* [Abadi et al. 1991]. In addition, $\mathrm{CC}_S$ does not reduce substitutions of expressions into functions.

Since $\mathrm{CC}_S$ extends CC, every CC expression is trivially a $\mathrm{CC}_S$ expression. We denote this trivial transformation from CC to $\mathrm{CC}_S$ as $\sigma$. Then, we define the defunctionalization transformation from

$\mathrm{CC}_S$ to DCC in a similar way as that from CC to DCC – an expression transformation $[\![-]\!]$ and a meta-function $[\![-]\!]_d$ for extracting definitions. Next, we show that $\sigma$ and defunctionalization for $\mathrm{CC}_S$ preserve reduction sequences and they commute with the transformation from CC into DCC. As a corollary, defunctionalization from CC to DCC preserves reduction sequences. In other words, we show that the following diagram commutes for all CC-expressions $M$ and $N$ (contexts omitted).




$\mathrm{CC}_S$ is an extension of CC with new syntax, type derivation rules, reduction rules, and equivalence rules (Fig. 5). We write $\mathrm{CC}_S$ expressions in a *teal, mathematical font* to avoid ambiguity. $\mathrm{CC}_S$ extends the CC syntax with *syntactic substitutions* of the form $M\{x \mapsto N\}$.

Type rules for variables, universes, Π-types, functions, and equivalence in $\mathrm{CC}_S$ are the same as the standard rules in CC, except that the type of an application $M\ N$ is $B\{x \mapsto N\}$ with the syntactic substitution. The type of a substitution $M\{x \mapsto N\}$ is the type of $M$ with $x$ substituted by $N$ (S-TY-SUBST).

$\mathrm{CC}_S$ has five reduction rules for substitutions, which are the standard meta-theoretic substitution rules for variables, universes, Π-types, and applications internalised into the language. Note that the meta-theoretic substitution in the CC's original beta-reduction rule $(\lambda x{:}A.M)\ N \triangleright M\{x \mapsto N\}$ is also replaced by the syntactic one. $\mathrm{CC}_S$ does not reduce substitutions into functions, but it $\beta$-reduces them when they are applied to arguments (S-RED-CLOSURE). We write $M\{x_1 \mapsto N_1, x_2 \mapsto N_2\}$ for a substitution followed by another substitution $(M\{x_1 \mapsto N_1\})\{x_2 \mapsto N_2\}$, and $M\{\overline{y} \mapsto \overline{N}\}$ for a sequence of substitutions $(((M\{y_1 \mapsto N_1\})\{x_2 \mapsto N_2\})\cdots)\{y_n \mapsto N_n\}$.

Like in CC, two terms in $\mathrm{CC}_S$ are equivalent if they $\beta$-reduce to the same expression or are $\eta$-equivalent. In addition, $\mathrm{CC}_S$ has two symmetric rules (S-EQ-CLOSURE1) and (S-EQ-CLOSURE2) for determining when a sequence of substitutions into a function $(\lambda x{:}A.M)\{\overline{y} \mapsto \overline{N}\}$ is equivalent to another expression. This is essentially a variant of the $\eta$-equivalence rules that is compatible with substitutions – $(\lambda x{:}A.M)\{\overline{y} \mapsto \overline{N}\}$ is equivalent to $N$ if applying $N$ to $x$ is equivalent to the function body $M$ with $\overline{y}$ being substituted for $\overline{N}$.

Now, we define the defunctionalization transformation from $\mathrm{CC}_S$ to DCC, which is the transformation from CC to DCC extended with the following two rules. We use $[\![-]\!]$ and $[\![-]\!]_d$ to stand for the expression transformation and the metafunction for extracting function definitions, and we apply the convention of tagging lambdas with unique identifiers $i$ ($i \in \mathbb{N}$) as usual.

$\boxed{\Gamma \vdash M : A \rightsquigarrow \mathsf{M}}$ *(Expression transformation)*

$$\frac{\Gamma, x : A \vdash M : B \rightsquigarrow \mathsf{M} \qquad \Gamma \vdash N : A \rightsquigarrow \mathsf{N}}{\Gamma \vdash M\{x \mapsto N\} : B\{x \mapsto N\} \rightsquigarrow \mathsf{M}[\mathsf{N}/\mathsf{x}]}\ \text{S-T-SUBST}$$

$\boxed{\Gamma \vdash M : A \rightsquigarrow_d \mathfrak{D}}$ *(Function extraction)*

$$\frac{\Gamma, x : A \vdash M : B \rightsquigarrow_d \mathfrak{D}_1 \qquad \Gamma \vdash N : A \rightsquigarrow_d \mathfrak{D}_2}{\Gamma \vdash M\{x \mapsto N\} : B\{x \mapsto N\} \rightsquigarrow_d \mathfrak{D}_1 \cup \mathfrak{D}_2}\ \text{S-D-SUBST}$$

The transformation turns a syntactic substitution in $\mathrm{CC}_S$ into a meta-theoretic substitution in DCC (S-T-SUBST); the function definitions in a substitution $M\{x \mapsto N\}$ are the union of the definitions in $M$ and $N$ (S-D-SUBST). Since substitutions into functions do not reduce in $\mathrm{CC}_S$, the

transformation from it into DCC have the following strong properties by definition, which are not true for the transformation from CC into DCC.

$$M \triangleright^* N \Longrightarrow [\![N]\!]_d \subseteq [\![M]\!]_d \tag{1}$$

$$[\![M\{x \mapsto N\}]\!] = [\![M]\!][[\![N]\!]/\mathtt{x}]. \tag{2}$$

Next, we show that the transformation preserves small step reductions in $\mathrm{CC}_S$ – if a $\mathrm{CC}_S$ program $M$ reduces to $N$ in one step, then the translated program $\mathtt{M}$ evaluates to $\mathtt{N}$ in a sequence.

Lemma 3.10 (Preservation of small step reductions). *If* $\Gamma \vdash M : A$ *and* $M \triangleright N$, *then* $[\![\Gamma]\!]_d \cup [\![M]\!]_d \vdash \mathtt{M} \triangleright^* \mathtt{N}$.

The transformation preserves sequences of reductions, and the proof follows from a trivial induction on the number of small steps in the sequence.

Lemma 3.11 (Preservation of reduction sequences ($\mathrm{CC}_S$)). *If* $\Gamma \vdash M : A$ *and* $M \triangleright^* N$, *then* $[\![\Gamma]\!]_d \cup [\![M]\!]_d \vdash \mathtt{M} \triangleright^* \mathtt{N}$.

The transformation is also coherent, i.e. it preserves the equivalence relation in $\mathrm{CC}_S$.

Lemma 3.12 (Coherence ($\mathrm{CC}_S$)). *If* $\Gamma \vdash M : A$, $\Gamma \vdash N : A$, *and* $\vdash M \equiv N$, *then* $\mathfrak{D} \vdash \mathtt{M} \equiv \mathtt{N}$, *where* $\mathfrak{D} = [\![\Gamma]\!]_d \cup [\![M]\!]_d \cup [\![N]\!]_d$.

Recall that $\sigma$ denotes the trivial transformation from CC to $\mathrm{CC}_S$. This trivial transformation commutes with the two term transformations by definition.

$$[\![\sigma(M)]\!] = [\![M]\!] \tag{3}$$

In addition, function definitions in $[\![\sigma(M)]\!]_d$ is a subset of the definitions in $[\![M]\!]_d$, because new function definitions appear in CC's type derivation trees as results of substitutions, but this does not happen in $\mathrm{CC}_S$.

$$[\![\sigma(M)]\!]_d \subseteq [\![M]\!]_d \tag{4}$$

We show that $\sigma$ also preserves sequences of reductions. As a convention, we write $M$ for $\sigma(M)$ when there is no ambiguity.

Lemma 3.13 (Preservation of reduction sequences ($\sigma$)). *If* $\Gamma \vdash M \triangleright^* N$, *then* $\Gamma \vdash M \triangleright^* M'$ *where* $\Gamma \vdash M' \equiv N$.

We can finally prove the preservation of reduction sequences for dependently typed defunctionalization (from CC to DCC) using the lemmas above.

Lemma 3.14 (Preservation of reduction sequences). *For all* $M$ *and* $N$, *if* $\Gamma \vdash M : A$ *and* $M \triangleright^* N$, *then we have*

$$\mathfrak{D}_\Gamma \cup \mathfrak{D}_M \cup \mathfrak{D}_N \vdash \mathtt{M} \triangleright^* \mathtt{M}', \tag{5}$$

$$\mathfrak{D}_\Gamma \cup \mathfrak{D}_M \cup \mathfrak{D}_N \vdash \mathtt{M}' \triangleright^* \mathtt{N} \tag{6}$$

*for some* $\mathtt{M}'$ *where* $(\mathfrak{D}_\Gamma \cup \mathfrak{D}_M \cup \mathfrak{D}_N) = ([\![\Gamma]\!]_d, [\![M]\!]_d, [\![N]\!]_d)$ *and* $(\mathtt{M}, \mathtt{N}) = ([\![M]\!], [\![N]\!])$ .

Since ground types and values do not contain functions, $[\![v]\!]_d = \cdot$, and the correctness of the transformation is just a special case of Lemma 3.14.

Corollary 3.15. *(Correctness) For all ground types* $A$ *and values* $v$ *of type* $A$,

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

The proof of type-preservation requires three lemmas: *substitution*, *preservation of reduction sequences*, and *coherence*. Lemma 3.14 established that dependently-typed defunctionalization preserves reduction sequences with the help of $\mathrm{CC}_S$, and now we prove the remaining two lemmas in a similar way. The substitution lemma states that defunctionalization is compatible with substitutions.

Lemma 3.16 (Substitution). *If* $\Gamma, x : A \vdash M : B$ *and* $\Gamma \vdash N : A$, *then* $\mathfrak{D} \vdash [[M[N/x]]] \equiv \mathsf{M}[\mathsf{N}/\mathsf{x}]$, *where* $\mathfrak{D} = \mathfrak{D}_\Gamma \cup \mathfrak{D}_M \cup \mathfrak{D}_N \cup \mathfrak{D}_M[N/x]$.

The coherence lemma states that defunctionalization is compatible with CC's coherence judgements.

Lemma 3.17 (Coherence). *If* $\Gamma \vdash M : A$, $\Gamma \vdash N : A$, *and* $\vdash M \equiv N$, *then* $\mathfrak{D} \vdash \mathsf{M} \equiv \mathsf{N}$, *where* $\mathfrak{D} = \mathfrak{D}_\Gamma \cup \mathfrak{D}_M \cup \mathfrak{D}_N$.

Finally, we show type preservation with an induction on CC's type derivation rules.

Theorem 3.18 (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* $(\mathfrak{D}_\Gamma, \mathfrak{D}_M) = ([[\Gamma]]_d, [[M]]_d)$ *and* $(\Gamma, \mathsf{M}, \mathsf{A}) = ([[\Gamma]], [[M]], [[A]])$.

### 3.5 Consistency and Type Safety of DCC

$\boxed{\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A} \rightsquigarrow_b M}$ *(Backward transformation)*

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

$$\frac{\mathfrak{D}; \Gamma \vdash \mathsf{A} : \mathsf{U_i} \rightsquigarrow_b A \quad \mathfrak{D}; \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{U_j} \rightsquigarrow_b B}{\mathfrak{D}; \Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{B} : \mathsf{U_{max(i,j)}} \rightsquigarrow_b \Pi x{:}A.B}\ \textsc{b-Pi}$$

$$\frac{\mathfrak{L}(\{\overline{\mathsf{x}} : \overline{\mathsf{A}}\}, \mathsf{x}{:}\mathsf{A} \mapsto \mathsf{M} : \mathsf{B}) \in \mathfrak{D} \quad \mathfrak{D}; \Gamma \vdash \overline{\mathsf{M}} : \overline{\mathsf{A}} \rightsquigarrow_b \overline{M} \quad \mathfrak{D}; \Gamma \vdash \mathsf{A} : \mathsf{U} \rightsquigarrow_b A \quad \mathfrak{D}; \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{M} : \mathsf{B} \rightsquigarrow_b M}{\mathfrak{D}; \Gamma \vdash \mathfrak{L}\{\overline{\mathsf{M}}\} : \Pi \mathsf{x}{:}\mathsf{A}[\overline{\mathsf{M}}/\overline{\mathsf{x}}].\mathsf{B}[\overline{\mathsf{M}}/\overline{\mathsf{x}}] \rightsquigarrow_b \lambda x{:}A[\overline{M}/\overline{x}].M[\overline{M}/\overline{x}]}\ \textsc{b-Label}$$

$$\frac{\mathfrak{D}; \Gamma \vdash \mathsf{L} : \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{B} \rightsquigarrow_b L \quad \mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A} \rightsquigarrow_b M}{\mathfrak{D}; \Gamma \vdash \mathsf{L}\,@\,\mathsf{M} : \mathsf{B}[\mathsf{M}/\mathsf{x}] \rightsquigarrow_b L\ M}\ \textsc{b-Apply} \qquad \frac{\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A} \rightsquigarrow_b M \quad \mathfrak{D}; \Gamma \vdash \mathsf{B} : \mathsf{U_i} \quad \mathfrak{D} \vdash \mathsf{A} \equiv \mathsf{B}}{\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{B} \rightsquigarrow_b M}\ \textsc{b-Equiv}$$

Fig. 6. Backward transformation

As a dependent type theory, DCC should be type-safe when it acts as a programming language and consistent when interpreted as a logic. Following Boulier et al. [2017] and Bowman and Ahmed [2018], we prove these properties in this section by defining a *backward transformation* from DCC to CC and showing that it preserves reduction sequences, so that reducing an expression in DCC is equivalent to reducing an expression in CC. The transformation is type-preserving and turns the logical interpretation of *false* in DCC into that of CC, so that valid proofs (i.e. well-typed programs) in DCC correspond to valid proofs in CC. This reduces the problem of proving the type safety and consistency of DCC to proving that of CC, which is a standard result [Coquand and Huet 1988]. In other words, we show that DCC can be modelled by CC in a consistent and meaning-preserving way. Type preservation for the backward transformation also requires the *substitution*, *preservation of reduction sequences*, and *coherence* lemmas, similar to the proof of Theorem 3.18, whose proofs are straightforward.

We define the backward transformation $[[-]]$ with a new judgement (Fig. 6) of the form $\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A} \rightsquigarrow_b M$ and $[[\mathsf{M}]] \triangleq M$. The translation maps variables, universes, Π-types, and applications back

to their corresponding forms in CC, and maps label expressions $\mathfrak{L}\{\overline{\mathsf{M}}\}$ where $\mathfrak{L}(\{\overline{\mathsf{x}} : \overline{\mathsf{A}}\}, \mathsf{x}{:}\mathsf{A} \mapsto \mathsf{M} : \mathsf{B}) \in \mathfrak{D}$ into $\lambda x{:}A[\overline{M}/\overline{x}].M[\overline{M}/\overline{x}]$ — a function with all of its free-variable values substituted in, where $A$, $M$, and $\overline{M}$ stand for $[\![\mathsf{A}]\!]$, $[\![\mathsf{M}]\!]$, and $[\![\overline{\mathsf{M}}]\!]$ respectively (B-Label). Intuitively, $[\![-]\!]$ decompiles a label back to the function it represents. The backward transformation also acts pointwise on type contexts.

In CC, the interpretation of the logical *false* is $\Pi x{:}U_0.x$. There is no closed expression with the *false* type. In DCC, the interpretation of *false* is $\Pi \mathsf{x}{:}\mathsf{U}_0.\mathsf{x}$, so the backward transformation preserves falseness by definition.

Next, we show that the backward transformation is compatible with substitutions. As a convention in this section, we write $M$ for $[\![\mathsf{M}]\!]$ when there is no ambiguity.

Lemma 3.19 (Backward transformation compatible with substitutions). *If* $\mathfrak{D}; \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{M} : \mathsf{B}$ *and* $\mathfrak{D}; \Gamma \vdash \mathsf{N} : \mathsf{A}$, *then* $[\![\mathsf{M}[\mathsf{N}/\mathsf{x}]]\!] = M[N/x]$.

Similar to proofs in Section §3.4, we show preservation of reduction sequences by showing that the transformation preserves small-step reductions. Using that, we show the coherence lemma for the backward transformation, and then the type preservation.

Lemma 3.20. *If* $\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A}$ *and* $\mathfrak{D} \vdash \mathsf{M} \triangleright^* \mathsf{N}$, *then* $M \triangleright^* N$.

Lemma 3.21. *If* $\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A}$, $\mathfrak{D}; \Gamma \vdash \mathsf{N} : \mathsf{A}$, *and* $\mathfrak{D} \vdash \mathsf{M} \equiv \mathsf{N}$, *then* $\vdash M \equiv N$.

Lemma 3.22. *If* $\mathfrak{D}; \Gamma \vdash \mathsf{M} : \mathsf{A}$, *then* $\Gamma \vdash M : A$.

As a corollary of Lemma 3.20 and Lemma 3.22, DCC is type-safe and consistent since CC is.

Theorem 3.23 (Type safety). *If* $\mathfrak{D}; \cdot \vdash \mathsf{M} : \mathsf{A}$, *then* $\mathfrak{D} \vdash \mathsf{M} \triangleright^* \mathsf{v}$ *for some irreducible value* $\mathsf{v}$.

That is, type safety guarantees that every well-typed closed DCC term reduces to a value in a finite number of steps.

Theorem 3.24 (Consistency). *There is no pair of a label context* $\mathfrak{D}$ *and DCC term* $\mathsf{M}$ *such that* $\mathfrak{D}; \cdot \vdash \mathsf{M} : \Pi \mathsf{A}{:}\mathsf{U}.\mathsf{A}$.

Interpreting DCC as a logic, the term $\Pi \mathsf{A}{:}\mathsf{U}.\mathsf{A}$ (which produces a term of any type $\mathsf{A}$) corresponds to *false*; it is backward-transformed to $\Pi A{:}U.A$, the representation of *false* in CC. Consistency of DCC means that there is no closed term of type $\Pi \mathsf{A}{:}\mathsf{U}.\mathsf{A}$; if there were, then translation would yield a corresponding term in CC, and CC would also be inconsistent.

## 4 RELATED WORK

*Type-Preserving Compilation.* Type-preserving compilation was initially developed for optimizing compilation and verifying the compiled code; it has been used extensively in compilers of simply-typed and polymorphic languages, and occasionally for dependently-typed languages. For example, Tarditi et al. [1996] present TIL (typed intermediate language), an ML compiler featuring type-directed code optimization of loops, garbage collections, and polymorphic function calls, and Morrisett et al. [1999] study a type-preserving translation from System F to the typed assembly language TAL. Xi and Harper [2001] later extended TAL to DTAL, an assembly language with a limited form of dependent types that serves as a compilation target for Dependent ML.

Guillemette and Monnier [2008] also present a type-preserving compiler from System F, but take a different approach, building typed intermediate representations using generalized algebraic data types and using the type system of the host language (GHC Haskell) to verify that each compiler phase preserves types. Embedding typed transformations in this way is a popular technique in the functional programming community, exemplified in work by Carette et al. [2009], which

presents type-preserving CPS transformations of an embedded language along with type-preserving optimizations based on partial evaluation.

Necula [1997]'s proof-carrying code is another early method for generating reliable executables. It relies on an external logical framework to check the correctness of proofs attached with the code.

Bowman and collaborators have developed several type-preserving translations for dependently-typed languages, including CPS transformation [Bowman et al. 2018], closure conversion [Bowman and Ahmed 2018] (building on typed closure conversion for System F by Minamide et al. [1996]), and translation to ANF [Koronkevich et al. 2022].

*Defunctionalization.* Defunctionalization was first presented by Reynolds [1972] as a programming technique to translate a higher-order interpreter into a first-order one [Reynolds 1972]. It has been used in a variety of applications, from ML compilers [Cejtin et al. 2000; Chin and Darlington 1996], to type-safe garbage collectors [Wang and Appel 2001], and encodings of higher-kinded polymorphism [Yallop and White 2014].

Defunctionalization was originally presented as an untyped translation. Using a family of monomorphic *apply* functions to make simply-typed defunctionalization type-preserving is a standard workaround in the literature [Bell et al. 1997; Cejtin et al. 2000; Nielsen 2000; Tolmach and Oliva 1998]. Danvy and Nielsen [2001] survey more examples of defunctionalization in practice.

Formalization of defunctionalization has up to this point focused on proving type preservation and correctness of the transformation. Bell et al. [1997] have shown that the translation for simply typed programs is type preserving. Nielsen [2000] has proved its partial correctness with denotational semantics, and Banerjee et al. [2001] have established total correctness using operational semantics. Pottier and Gauthier [2004] have formalized type-preserving polymorphic defunctionalization in System F extended with GADTs.

*Closure Conversion.* Like defunctionalization, *closure conversion* transformations also involve representing a closure as a first-order value that pairs a kind of code identifier with a collection of free variables. The formulations of closure conversion in the work referenced above [Bowman and Ahmed 2018; Minamide et al. 1996] differ markedly from defunctionalization: while defunctionalization involves a globally-defined map indexed by code identifiers (such as an *apply* function or our label environment), these closure conversions instead locally transform functions into code-and-environment pairs that can then be applied using a standard elimination rule. However, other formulations of closure conversion [e.g. Appel 1992; Siek 2012] additionally lift functions to top-level, making the transformation more similar to defunctionalization.

Closure conversion plays a key role in compilers for many functional languages, including Scheme [Steele Jr 1978], CAML [Mauny and Suárez 1986], Standard ML [Cejtin et al. 2000], Haskell [Leshchinskiy et al. 2006] and others. Recent work has focused on establishing sophisticated semantic properties, such as correctness of closure conversion in the presence of mutable state and control effects (even when linked with foreign-language code) [Mates et al. 2019], and preservation of time and space properties [Paraskevopoulou and Appel 2019].

*Refunctionalization.* Our backward translation (See §3.5) is related to *refunctionalization* [Danvy and Millikin 2009], the left-inverse of defunctionalization. As in refunctionalization, we replace target applications $\mathsf{M} \mathbin{@} \mathsf{N}$ with source applications $M\ N$, and labels $\mathfrak{L}\{\overline{\mathsf{M}}\}$ with abstractions $(\lambda x{:}A.M)[\overline{M}/\overline{x}]$ based on their implementations $\mathfrak{L}(\{\overline{\mathsf{x}} : \overline{\mathsf{A}}\}, \mathsf{x}{:}\mathsf{A} \mapsto \mathsf{M} : \mathsf{B})$ in the label context.

## ACKNOWLEDGMENTS

We thank David Sheets, András Kovács, Marcelo Fiore, and the anonymous reviewers for helpful comments, and Andreas Rossberg for shepherding the paper.

## IMPLEMENTATION

We provide a portable standalone implementation [Huang and Yallop 2023b] of the defunctionalization translation of §3, written in OCaml and compiled to run in a web browser using `js_of_ocaml` [Vouillon and Balat 2014]. The implementation performs type checking of CC (§3.1) and DCC (§3.2) terms, abstract defunctionalization (§3.3) and backwards translation from DCC to CC (§3.5), allowing the interested reader to experiment with the effects of the translation on real examples. We include several ready-made examples, including dependent composition, dependent pairs and finite sets.

## REFERENCES


Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. Explicit Substitutions. *J. Funct. Program.* 1, 4 (1991), 375–416. https://doi.org/10.1017/S0956796800000186

Benedikt Ahrens, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. 2018. Categorical structures for type theory in univalent foundations. *Log. Methods Comput. Sci.* 14, 3 (2018). https://doi.org/10.23638/LMCS-14(3:18)2018

Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press.

Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 2001. Design and Correctness of Program Transformations Based on Control-Flow Analysis. In *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2215)*, Naoki Kobayashi and Benjamin C. Pierce (Eds.). Springer, 420–447. https://doi.org/10.1007/3-540-45500-0_21

Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. 1999. CPS Translations and Applications: The Cube and Beyond. *High. Order Symb. Comput.* 12, 2 (1999), 125–170. https://doi.org/10.1023/A:1010000206149

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

Jeffrey M. Bell, Françoise Bellegarde, and James Hook. 1997. Type-Driven Defunctionalization. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman (Eds.). ACM, 25–37. https://doi.org/10.1145/258948.258953

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 182–194. https://doi.org/10.1145/3018610.3018620

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

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

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

Edwin C. Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Bernd Braßel. 2011. *Implementing Functional Logic Programs by Translation into Purely Functional Programs.* Ph. D. Dissertation. https://macau.uni-kiel.de/receive/diss_mods_00007056

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-Directed Closure Conversion for Typed Languages. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1782)*, Gert Smolka (Ed.). Springer, 56–71. https://doi.org/10.1007/3-540-46425-5_4

Wei-Ngan Chin and John Darlington. 1996. A Higher-Order Removal Method. *LISP Symb. Comput.* 9, 4 (1996), 287–322.

The Coq Development Team. 2022. The Coq Reference Manual. https://coq.inria.fr/distrib/current/refman/.

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

Olivier Danvy and Kevin Millikin. 2009. Refunctionalization at work. *Sci. Comput. Program.* 74, 8 (2009), 534–549. https://doi.org/10.1016/j.scico.2007.10.007

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and practice of declarative programming, September 5-7, 2001, Florence, Italy*. ACM, 162–174. https://doi.org/10.1145/773184.773202

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

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

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

Yulong Huang and Jeremy Yallop. 2023a. Defunctionalization with Dependent Types. arXiv:2304.04574 [cs.PL]

Yulong Huang and Jeremy Yallop. 2023b. Defunctionalization with Dependent Types: Artifact. https://doi.org/10.5281/zenodo.7709681.

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

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

Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. 2006. Higher Order Flattening. In *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 3992)*, Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra (Eds.). Springer, 920–928. https://doi.org/10.1007/11758525_122

Zhaohui Luo. 1990. *An extended calculus of constructions.* Ph. D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/12487

Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 16:1–16:15. https://doi.org/10.1145/3354166.3354181

Michel Mauny and Ascánder Suárez. 1986. Implementing Functional Languages in the Categorical Abstract Machine. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986*, William L. Scherlis, John H. Williams, and Richard P. Gabriel (Eds.). ACM, 266–278. https://doi.org/10.1145/319838.319869

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

J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (1999), 527–568. https://doi.org/10.1145/319301.319345

George C. Necula. 1997. Proof-Carrying Code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 106–119. https://doi.org/10.1145/263699.263712

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

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures (Lecture Notes in Computer Science, Vol. 5832)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.). Springer, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP (2019), 83:1–83:29. https://doi.org/10.1145/3341687

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

Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. 2005. Continuations from generalized stack inspection. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 216–227. https://doi.org/10.1145/1086365.1086393

Peter Podlovics, Csaba Hruska, and Andor Pénzes. 2021. A Modern Look at GRIN, an Optimizing Functional Language Back End. *Acta Cybernetica* (Feb. 2021). https://doi.org/10.14232/actacyb.282969

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

François Pottier and Nadji Gauthier. 2006. Polymorphic typed defunctionalization and concretization. *High. Order Symb. Comput.* 19, 1 (2006), 125–162. https://doi.org/10.1007/s10990-006-8611-7

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, John J. Donovan and Rosemary Shields (Eds.). ACM, 717–740. https://doi.org/10.1145/800194.805852

Jeremy Siek. 2012. The Essence of Closure Conversion. http://siek.blogspot.com/2012/07/essence-of-closure-conversion.html.

Guy Lewis Steele Jr. 1978. *RABBIT: A Compiler for SCHEME.* Master's thesis. MIT.

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

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

Amin Timany and Matthieu Sozeau. 2017. Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC). *CoRR* abs/1710.03912 (2017). arXiv:1710.03912 http://arxiv.org/abs/1710.03912

Andrew P. Tolmach and Dino Oliva. 1998. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *J. Funct. Program.* 8, 4 (1998), 367–412. http://journals.cambridge.org/action/displayAbstract?aid=44181

Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Softw. Pract. Exp.* 44, 8 (2014), 951–972. https://doi.org/10.1002/spe.2187

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

Daniel C. Wang and Andrew W. Appel. 2001. Type-preserving garbage collectors. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 166–178. https://doi.org/10.1145/360204.360218

Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1159876.1159877

Stephanie Weirich and Chris Casinghino. 2010. Generic Programming with Dependent Types. In *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures (Lecture Notes in Computer Science, Vol. 7470)*, Jeremy Gibbons (Ed.). Springer, 217–258. https://doi.org/10.1007/978-3-642-32202-0_5

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

Jeremy Yallop and Leo White. 2014. Lightweight Higher-Kinded Polymorphism. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 119–135. https://doi.org/10.1007/978-3-319-07151-0_8

Received 2022-11-10; accepted 2023-03-31