Defunctionalization with dependent types

Yulong Huang
Robinson College

UNIVERSITY OF CAMBRIDGE

Submitted in partial fulfillment of the requirements for the Computer Science Tripos, Part III

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: yh419@cam.ac.uk

May 27, 2022

Declaration

I, Yulong Huang of Robinson College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 9487

Signed: Yulong Huang

Date: May 27, 2022

Abstract

This dissertation studies *defunctionalization*, a program transformation that turns a higher-order functional program into a first-order one. Type-preserving defunctionalization for simply-typed and polymorphic systems is well-studied in the literature, and my work extends defunctionalization further to dependently-typed systems.

I illustrate that Pottier and Gauthier's polymorphic defunctionalization does not extend to dependently-typed languages. Then, I present *abstract defunctionalization* as an alternative approach. Abstract defunctionalization consists of a target language with a primitive notion of function labels that fits the abstract description of defunctionalization, and a transformation from the source language to the target language. I prove the transformation type-preserving and correct, and I show that the target language is type-safe and consistent. An interpreter of the target language and the transformation are implemented in OCaml.

Acknowledgements

I would like to thank my project supervisor, Dr Jeremy Yallop, and my Director of Studies, Prof Alan Mycroft, for their countless valuable suggestions, feedback and comments on my project, writing, and academic study in general. I also thank my project co-supervisor David Sheets for sharing his ideas and insights in our weekly meetings.

Contents

List of Figures

Chapter 1

Introduction

Dependent types are used to verify the correctness of large-scale programs, as they can express program specifications with their type system and guarantee that all specifications hold by type-checking. A modern trend in compiling dependently-typed languages is to turn source programs into target programs in a dependently-typed intermediate language and preserve the type information [1]. In this way, program specifications are preserved through types and checking the correctness of separately compiled and linked programs becomes possible. Type-preserving compilation is well-studied in non-dependently-typed scenarios, and the current challenge is adapting compiler transformations for non-dependent types to dependent types.

This dissertation studies *defunctionalization*, a program transformation that turns a higher-order functional program into a first-order one by replacing every function with a *label* (a special kind of first-class value). Type-preserving defunctionalization for simply-typed and polymorphic systems is well-established in the literature [2, 3], and my work extends defunctionalization further to dependently-typed systems.

This introduction motivates the development of type-preserving compilers for dependently-typed languages. It then summarizes the contributions of this dissertation and discusses related works in type-preserving compilation.

1.1 Motivation and contributions

Dependently-typed systems are used for verifying the functional correctness of a range of large-scale software, from the CompCert C compiler [4] to the CertiKOS OS kernel [5]. In dependently-typed systems, types may contain term variables and the type of a function's output could depend on the *value* of its input. The type of a *dependent function* is denoted as $\Pi x : A.\, B$, where B is called an *indexed family* of types (indexed by terms of type A). With the *proposition-as-types* principle, a dependently-typed calculus is both a programming language and a logic. So, it is possible to specify invariants and Hoare-logic style pre- and post-conditions for programs as types, and dependent type-checking guarantees that the specified conditions hold. For example, I can specify the following type for a safe natural-number division function in Agda, which takes arguments `x`, `y`, and an proof that `y` is non-zero to type-check.

```
(* Returns x / y *)
div : (x : Nat) -> (y : Nat) -> (p : y > 0) -> Nat
```

However, correctness is not guaranteed for separately compiled programs. For example, programmers may break specifications by linking incompatible pieces of separately compiled code. There is no way to check the specifications of separately compiled and linked programs, since the current implementation of compilers for dependently-typed languages removes type information after compiling. In other words, there is no way to enforce correctness on the target code with dependent types and type-checking.

This problem can be solved with type-preserving compilation, a well-known technique in non-dependently-typed scenarios. These compilers preserve function behaviours and program semantics like regular compilers. Furthermore, they preserve the type information – well-typed source programs are turned into well-typed target programs in a typed intermediate language. It is possible to type-check the target code and ensure correctness statically in separately compiled and linked programs using the preserved types. Type information is only removed after the program is fully linked and checked. The current challenge of developing type-preserving compilers for dependently-typed languages is adapting non-dependently-typed compiler transformations to dependent types.

Defunctionalization is a compilation technique that turns higher-order functional programs into first-order ones. It is well-studied in simply-typed and polymorphic scenarios, and my work extends defunctionalization further to dependently-typed systems.

Contributions The main contribution of this dissertation is *abstract defunctionalization*, a method of performing type-preserving defunctionalization with dependent types. Abstract transformation consists of a target language with a primitive notion of function *labels* that fits the abstract description of defunctionalization, and a transformation from the source language into the target language. I prove that the transformation is type-preserving and correct, and the target language is type-safe and consistent.

The remainder of this dissertation is structured as follows. In Chapter 2, I explain the standard method of defunctionalization for simply-typed and polymorphic programs and defines the source language of dependently-typed defunctionalization. In Chapter 3, I illustrate that Pottier and Gauthier's standard technique of polymorphic defunctionalization [3] does not extend to dependently-typed systems.

In Chapter 4, I present abstract defunctionalization as an alternative method. This chapter formally presents the Defunctionalized Calculus of Constructions (DCC), the target language of abstract defunctionalization. It also defines the defunctionalization transformation from the source language into DCC. Chapter 5 presents the proof of type preservation and correctness of the transformation and shows that DCC is consistent and type-safe. Chapter 6 concludes the dissertation with a closing remark on implementing DCC in OCaml and future work.

1.2 Related work in type-preserving compilation

Type-preserving compilation was initially developed for optimizing and verifying the compiled code, and it is a well-known technique in compiler design for languages that are not dependently-typed. Tarditi et al. [6] developed TIL (typed intermediate language), an ML compiler featuring type-directed code optimization of loops, garbage collections, and polymorphic function calls.

Morrisett et al. [7] developed a type-preserving translation from System F to TAL (typed assembly language). Later, Xi and Harper [8] introduced DTAL, which extends TAL with a restricted form of dependent types, making it suitable for compiling a variant of ML with dependent datatype extensions. Necula's proof-carrying code [9] 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.

Recently, Bowman et al. contributed a series of work about compiling with dependent types, including the type-preserving continuation-passing style (CPS) and closure conversion transformations [10, 11].

Chapter 2

Background

This chapter gives a survey about defunctionalization transformation in non-dependently-typed systems (Section 2.1) and presents the definition the source language (CC) for dependently-typed defunctionalization (Section 2.2).

2.1 Defunctionalization

Defunctionalization is a whole-program transformation that turns higher-order functional programs into first-order ones. This section explains type-preserving defunctionalization for simply-typed and polymorphic source programs with illustrating examples.

2.1.1 Simply-typed programs

Defunctionalization is based on the observation that a program has only finitely many function definitions. Therefore, if the whole program is available, we can enumerate every function and replace them with distinct constructors of an *algebraic data type (ADT)* in the target language. I refer to these constructors as *labels*. Defunctionalization translates function applications into calls to an auxiliary *apply* function, which takes a label and the function argument, performs case analysis on the label, and returns the result of the corresponding function applied to the argument. In other words, letting $[\![-]\!]$ denote the transformation, $[\![e_1\ e_2]\!] = \mathit{apply}\ [\![e_1]\!]\ [\![e_2]\!]$, where $\mathit{apply}\ [\![e_1]\!]\ [\![e_2]\!]$ simulates the behavior of $e_1\ e_2$ in the source language.

Suppose the source and the target language are both OCaml. Example 2.1.1 shows how to defunctionalize a simply typed higher-order program to a simply-typed first-order program.

Example 2.1.1. Let the source program be:

```
let rec map : (int -> int) -> int list -> int list =
    fun f ls -> match ls with
```

```
        | [] -> []
        | hd :: tl -> (f hd) :: (map f tl)
in
    map (fun x -> x + 1) (map (fun x:int -> x) [1])
```

It contains two first-class anonymous functions of the type `int -> int`. Defunctionalization involves defining a data type `lam` with two constructors, one for each function.

```
(* Incr represents (fun x -> x + 1) *)
(* Id represents (fun x:int -> x) *)
type lam = Incr | Id
```

The next step is defining the auxiliary function *apply*: `apply f arg` returns the result of applying the source-program function which `f` represents to the argument `arg`. If `f` is `Incr`, it returns `arg + 1`, and similarly for case `Id`.

```
let apply : lam -> int -> int =
    fun f arg -> match f with
        | Incr -> arg + 1
        | Id -> arg
```

Now, the defunctionalized target program can be obtained by replacing all the first-class functions with their corresponding labels and all the function applications with calls to `apply`.

```
let rec map' : lam -> int list -> int list =
    fun f ls -> match ls with
        | [] -> []
        | hd :: tl -> (apply f hd) :: (map' f tl)
in
    map' Incr (map' Id [1])
```

Note that `map'` now takes first-order values of type `Lam` instead of functions.

Type preservation

Informally speaking, a type-preserving transformation turns a well-typed source program to a well-typed target program. In Example 2.1.1, all first-class functions have the same type. Example 2.1.2 shows that defunctionalization produces ill-typed programs when first-class functions are not all in the same type.

Example 2.1.2. Let the source program be:

```
let compose : (int -> bool) -> (int -> int) -> int -> bool =
    fun g f x -> g (f x)
in
    compose (fun x -> x > 0) (fun x:int -> x) 1
```

Data type lam and the apply function are defined similarly as in Example 2.1.1.

```
(* ToBool represents (fun x -> x > 0) *)
(* Id represents (fun x : int -> x) *)
type lam = ToBool | Id

let apply f arg = match f with
    | ToBool -> arg > 0
    | Id -> arg
```

Function apply is ill-typed – it returns a bool in case ToBool and returns an int in case Id.

A minor adjustment can fix this problem: defining two apply functions, one for functions of type int -> bool, another one for functions of type int -> int.

```
(* apply_int_bool : Lam -> int -> bool *)
let apply_int_bool f arg = match f with
        | ToBool -> arg > 0


(* apply_int_int : Lam -> int -> int *)
let apply_int_int f arg = match f with
        | Id -> arg

let compose' g f x = apply_int_bool g (apply_int_int f x) in
    compose' ToBool Id 1
```

In general, simply typed defunctionalization need to define a family of monomorphic apply functions (i.e. one specialized function apply_a_b for each function type a -> b in the source program) to achieve type preservation.

2.1.2 Polymorphic programs

In languages with polymorphism and *generalized algebraic data types (GADTs)*, defunctionalization can be done similarly as in simply-typed scenarios – first-class functions are replaced by labels, and applications are translated to calls into the *apply* function. The difference is that we only need one polymorphic *apply*, and function labels are defined as constructors of a GADT. GADTs allow data types to take type parameters, and constructors may return specific instantiations of the parameters. Suppose our target language now supports GADTs, Example 2.1.3 shows how to defunctionalize a polymorphic higher-order program.

Example 2.1.3. Consider the polymorphic composition function:

```
let compose : type a b c. (b -> c) -> (a -> b) -> a -> c =
    fun g f x -> g (f x)
```

```
in
    compose (fun x -> x > 0) (fun x -> x) 1
```

`(fun x -> x)` is the polymorphic identity function. The type `lam` takes two type parameters, and a function of type `a -> b` in the source program translates into a label of type `(a, b) lam`. Note that the constructor `ToBool`, which corresponds to a non-polymorphic function, has instantiated parameters.

```
type ('a, 'b) lam =
      ToBool : (int, bool) lam
    | Id : ('a, 'a) lam

let apply : type a b. (a, b) lam -> a -> b =
    fun f arg -> match f with
        | ToBool -> arg > 0
        | Id -> arg
```

Again, the target program is obtained by replacing first-class functions with their corresponding labels and function applications with calls to `apply`, and the result is well-typed.

```
let compose' : type a b c.
                 (b, c) lam -> (a, b) lam -> a -> c =
    fun g f x -> apply g (apply f x)
in
    compose' ToBool Id 1
```

Functions with free variables

Defunctionalization was initially presented as a programming technique for eliminating functions with functional arguments [12]. The concept and technique generalize to eliminating functions that return functions, for example, the curried version of integer addition, whose type is `int -> (int -> int)`. The returned functions may contain free variables, and defunctionalization translates free variables into arguments to their corresponding function labels. In general, the label of a function `f:a -> b` with free variables `x1:t1, ..., xn:tn` has the form:

```
Fun : t1 -> ... -> tn -> (a, b) lam
```

Arguments to the label form an *environment* that stores values of free variables. A label with fully-applied arguments is similar to a *closure*, except that a closure stores the function pointer together with the environment, whereas defunctionalization keeps the function definition separately in *apply*. Example 2.1.4 shows the defunctionalization of the curried integer addition.

Example 2.1.4. The source program is the curried integer addition.

```
let plus : int -> (int -> int) =
    fun x -> (fun y -> x + y)
in
    plus 1 2
```

There are two function definitions in the code above. Applying the polymorphic defunctionalization technique explained in Example 2.1.3, `lam` and `apply` are defined as follows.

```
type ('a, 'b) lam =
      Plus : (int, (int, int) lam) lam
    | Plus_x : int -> (int, int) lam

let apply : type a b. (a, b) lam -> a -> b =
    fun f arg -> match f with
        | Plus -> Plus_x arg
        | (Plus_x x) -> x + arg
```

There is a free variable `x` of type `int` in `(fun y -> x + y)`, so the corresponding constructor `Plus_x` takes an argument of type `int`.

The target program is obtained in a similar way as in previous examples.

```
let plus' : (int, (int, int) lam) lam = Plus
in apply (apply plus' 1) 2
```

2.1.3 Related work in defunctionalization

Defunctionalization is first presented in its untyped form by Reynolds in the 1970s as a programming technique to translate a higher-order interpreter into a first-order one [12]. It is later used as an implementation technique in various applications such as ML compilers [13, 14], type-safe garbage collectors [15], and representing higher-kinded polymorphism in OCaml [16]. The method of using a family of monomorphic *apply* functions to achieve type preservation is the standard workaround for simply-typed defunctionalization in the literature [2, 17, 14, 18]. Danvy and Nielsen's survey contains more examples of defunctionalization in practice [19].

Formalization of defunctionalization focused on proving type preservation and correctness of the transformation. Bell et al. proved that the transformation for simply typed programs is type preserving [2]. Nielsen presented a proof for its partial correctness in denotational semantics [18], and Banerjee et al. provided a proof for total correctness in operational semantics [20]. Pottier and Gauthier formalized type-preserving polymorphic defunctionalization in System F extended with GADTs [3].

2.2 The source language (CC)

The source language (which I refer to as CC) I will use to define dependently-typed defunctionalization is a variant of Coquand's Calculus of Constructions [21]. To be precise, it is a subset of Luo's Extended Calculus of Constructions [22], which combines Coquand's original Calculus of Constructions (with only one impredicative universe) with a Martin-Löf style universe hierarchy[1].

Dependently-typed languages have *universes*, also known as *kinds* or *sorts*, and they are essentially the types of types. CC uses an infinite hierarchy of predicative universes U_i, where i ranges over natural numbers, and the type of universe U_i is U_{i+1}. Therefore, universes form an infinite hierarchy:

$$U_0 : U_1 : U_2 : \cdots$$

CC's syntax is presented in Figure 2.1. Term expressions of CC are: variables (from an infinite set of variable names), universes, dependent function types (or Π-types), applications, and functions. A CC context is a list of variable-expression pairs.

CC's typing judgements are of the form $\Gamma \vdash e : A$, where Γ is a context and e, A are terms. A context is *well-formed* if every variable in it is associated with a valid type, that is, the associated expression's type is a universe. The judgement for the well-formedness of contexts is $\vdash \Gamma$.

CC's rules for typing (Fig 2.2) and well-formedness of contexts (Fig 2.3) are mutual-inductively defined. The type of a variable x is A if $x : A$ is present its well-formed context Γ (Var). The type of a universe U_i is U_{i+1} (Universe), and the type of $\Pi x : A.\, B$ is the highest universe among universes of A and B (Pi). If e has type B in some context Γ extended with $x : A$, then $\lambda x : A.\ e$ has the dependent function type $\Pi x : A.\, B$ (Lambda). Applications have types $B[e_2/x]$, since the output of a function type may depend on the input value (Apply). If an expression e has type A and A is equivalent to B (under context Γ), then e also has type B (Equiv).

I write $\Gamma \vdash A : U$ to mean that $\Gamma \vdash A : U_i$ for some i (which means that A is a type), and $A \to B$ stands for the Π-type $\Pi x : A.\, B$ where B does not depend on x. Base types are omitted for simplicity, but I will use base types like the unit type and the natural numbers freely in examples.

Figure 2.4 defines reduction and equivalence in CC. The only reduction rule is the β-reduction, $(\lambda x : A.\ e_1)\ e_2 \ \triangleright\ e_1[e_2/x]$. I write $e_1 \triangleright^* e_2$ to mean that e_1 reduces to e_2 in a

[1]A definition is impredicative if it is self-referencing. The type of an impredicative universe is itself, like `Type : Type`. In a predicative hierarchy of universes, the type of a universe U_i is always a larger universe U_{i+1}.

$$\begin{array}{llcl}
\textit{Universes} & U & ::= & U_i \\
\textit{Expressions} & e,\, A,\, B & ::= & x \mid U \mid \Pi x:A.\, B \mid e_1\; e_2 \mid \lambda x:A.\; e \\
\textit{Context} & \Gamma & ::= & \cdot \mid \Gamma,\, x:A
\end{array}$$

Figure 2.1: CC Syntax

$$\frac{x:A \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x:A} \quad \text{(Var)}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash U_i : U_{i+1}} \quad \text{(Universe)}$$

$$\frac{\Gamma \vdash A:U_i \qquad \Gamma, x:A \vdash B:U_j}{\Gamma \vdash \Pi x:A.\, B : U_{max(i,j)}} \quad \text{(Pi)}$$

$$\frac{\Gamma \vdash e_1 : \Pi x:A.\, B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\; e_2 : B[e2/x]} \quad \text{(Apply)}$$

$$\frac{\Gamma, x:A \vdash e:B}{\Gamma \vdash \lambda x:A.\; e : \Pi x:A.\, B} \quad \text{(Lambda)}$$

$$\frac{\Gamma \vdash e:A \qquad \Gamma \vdash B:U \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash e:B} \quad \text{(Equiv)}$$

Figure 2.2: CC Typing

sequence with zero or more steps. The reduction rule can be thought of the operational semantics of CC.

In CC, two terms are equivalent if they reduce to the same term (Eq-reduce) or are η-equivalent. The η-equivalence is defined by two symmetric rules (Eq-Eta1) and (Eq-Eta2). Rule (Eq-Eta1) states that e_1 and e_2 are equivalent if e_1 reduces to a lambda abstraction $\lambda x:A.\; B$, e_2 reduces to some e_2', and $e_2'\; x \equiv e$ in the context extended with $x:A$.

$$\frac{}{\vdash \cdot} \quad \text{(WF-Empty)}$$

$$\frac{\vdash \cdot \qquad \Gamma \vdash A:U}{\vdash \Gamma, x:A} \quad \text{(WF-Con)}$$

Figure 2.3: Well-formedness of CC context

$$(\lambda x{:}A.\ e_1)\ e_2 \ \rhd\ [e_1/e_2] \qquad \text{(Beta)}$$

$$\frac{e_1 \rhd^* e \qquad e_2 \rhd^* e}{\Gamma \vdash e_1 \equiv e_2} \qquad \text{(Eq-reduce)}$$

$$\frac{e_1 \rhd^* \lambda x{:}A.\ e \qquad e_2 \rhd^* e_2' \qquad \Gamma, x{:}A \vdash e \equiv e_2'\ x}{\Gamma \vdash e_1 \equiv e_2} \qquad \text{(Eq-Eta1)}$$

$$\frac{e_1 \rhd^* e_1' \qquad e_2 \rhd^* \lambda x{:}A.\ e \qquad \Gamma, x{:}A \vdash e_1'\ x \equiv e}{\Gamma \vdash e_1 \equiv e_2} \qquad \text{(Eq-Eta2)}$$

Figure 2.4: CC reduction and equivalence

CC has a useful property: if $\Gamma \vdash e{:}A$, then $\Gamma \vdash A{:}U$. Precisely, in the derivation tree of $\Gamma \vdash e{:}A$, there must be a sub-derivation tree of $\Gamma \vdash A{:}U$.

As standard results from the literature [21, 22], CC is type safe and consistent, and type-checking in CC is decidable. Despite having a minimalistic syntax, CC can express many useful functions, for example, the polymorphic identity function and the dependent composition function.

Example 2.2.1. In System F, the polymorphic identity function has type $\forall a.\ a \to a$. In CC, we can define a dependent function that takes a type A and returns the identity function of type $A \to A$.

$$\begin{aligned} \mathit{identity} &\triangleq \lambda A{:}U_0.\ \lambda x{:}A.\ x \\ \cdot \vdash \mathit{identity} &: \Pi A{:}U_0.\ A \to A \end{aligned}$$

Example 2.2.2. This example shows that CC can write complicated but meaningful programs like the dependent composition function. The usual composition takes a function f of type $A \to B$, a function g of type $B \to C$, an input x of type A (here A, B, and C are simple types) and returns the result of $g(f\ x)$. The dependent composition generalizes this to the case where B is a family of types indexed by terms of type A, and C is a family indexed by terms x of type A and terms of type $B\ x$.

$$\begin{aligned} &\mathit{compose} \triangleq \lambda g{:}(\Pi x{:}A.\, \Pi y{:}B\ x.\, C\ x\ y).\ \lambda f{:}(\Pi x{:}A.\, B\ x).\ \lambda x{:}A.\ g\ x\ (f\ x) \\ &\qquad \Gamma \triangleq \cdot,\ A{:}U_0,\ B{:}A \to U_0,\ C{:}\Pi x{:}A.\,(B\ x) \to U_0 \\ &\qquad \Gamma \vdash \mathit{compose} : \Pi g{:}\_.\, \Pi f{:}\_.\, \Pi x{:}A.\, C\ x\ (f\ x) \end{aligned}$$

Chapter 3

A failing attempt

This chapter shows that Pottier and Gauthier's polymorphic defunctionalization technique cannot be adapted to dependent types. Section 3.1 introduces *inductive families*, a more general version of GADTs in dependently-typed languages. Section 3.2 proves that it is impossible to generalize polymorphic defunctionalization with GADTs to dependently-typed defunctionalization with inductive families.

3.1 Inductive families

This section introduces inductively defined data types in dependent types (inductive families), and I present code examples in Agda, a dependently-typed programming language with a Haskell-like syntax. No knowledge of Agda is required to understand the contents of this section, except for two things:

- In Agda, there is an infinite hierarchy of predicative universes

  $$\texttt{Set}_0 : \texttt{Set}_1 : \texttt{Set}_2 : \cdots$$

  similar to the hierarchy in CC.
- The dependent function type $\Pi x : A.\, B$ is written as `(x:A) -> B` in Agda's syntax.

Agda supports inductive families, which are essentially inductively defined data types indexed by terms [23]. Their constructors may return elements in an arbitrary type of the indexed family. Example 3.1.1 shows the definition of `Fin`, the inductive family of finite sets.

Example 3.1.1. `Fin` is a data type indexed by natural numbers (`Nat`) such that there are exactly `n` valid expressions of type `Fin n`. Given a natural number `n`, `fzero` constructs an element of type `Fin n+1`; given a natural number `n` and an element of type `Fin n`, `fsuc` constructs an element of type `Fin n+1`.

```
data Fin : Nat -> Set₀ where
    fzero : (n : Nat) -> Fin (n + 1)
    fsuc : (n : Nat) -> Fin n -> Fin (n + 1)
```

GADTs can be considered as a limited version of inductive families that can only be indexed by types. In general, inductive families are defined in the following form.

```
data D : (y₁ : T₁) -> ··· -> (yₙ : Tₙ) -> Set_d where
    c₁ : A₁
    c₂ : A₂
    ...
```

D is an inductive family in $\mathtt{Set}_d$ indexed by terms of type $T_1, \cdots, T_n$. For each constructor c_i, it takes a number of arguments $(z_i:S_i)$ and constructs an element of type $(D\ t_1\ \cdots\ t_n)$, where each t_i is an expression of type T_i. Concretely, each A_i takes the form of

$(z_1\ :\ S_1)\ \texttt{->}\ \cdots\ \texttt{->}\ (z_m\ :\ S_m)\ \texttt{->}\ (D\ t_1\ \cdots\ t_n).$

Note that arguments to D and c are dependently typed – T_{i+1} could mention $y_1 \cdots y_i$, and S_{i+1} could mention $z_1 \cdots z_i$. Also, $y_1 \cdots y_n$ are not bound in A_i.

Not all inductive definitions are consistent – some contain impredicative constructors (the constructor's universe is larger than the data type's), and some admit non-terminating functions. These definitions are sources of paradoxes, which Agda excludes by performing a *universe check* and a *positivity check* on inductive definitions. Take the data type D defined above as an example, it must satisfy two additional conditions:

- Types of its constructors must fit in the universe of D.
- D must occur *strictly positively* in types of its constructor's arguments.

Concretely, the universe of every argument type S_i (the type of S_i) should be smaller than $\mathtt{Set}_d$ to pass the universe check. Having strict positivity means that constructors cannot return self-indexing types like (D (D ···) ···), and every argument type S_i must either:

- not mention D at all.
- mention D in the form of $S_i\ =\ (x_1{:}C_1)\ \texttt{->}\ \cdots\ \texttt{->}\ (x_k{:}C_k)\ \texttt{->}\ D$, where D does not occur in any C_j (arguments to D omitted).

The requirements for strict posivitity and universes are shared across many dependently-typed languages that support inductive families, like Coq's Gallina [24] or Timany and Sozeau's pCuIC [25].

3.2 Failure to defunctionalize

In this section, I illustrate that it is impossible to adapt the method of performing polymorphic defunctionalization to dependently-typed languages using inductive families. Chapter 2 illustrated that type-preserving defunctionalization can be done with ADTs for simple types and GADTs for polymorphic types. In polymorphic defunctionalization, a function of type `a -> b` is translated to a constructor of a GADT `lam`, and the constructor's return type is `(a, b) lam` (see Example 2.1.3). Similarly, we can try to define an inductive family `Pi`, and translate each dependent function of type $\Pi x : A.\, B$ to a constructor of `Pi A B` (suppose that `Pi` has a way of encoding `B`'s dependency on terms of type `A`).

Suppose the source language is CC and the target language is Agda. Let $[\![-]\!]$ denote the defunctionalization transformation. I assume that $[\![\lambda x : A.\ e]\!] = \texttt{Fun}$, i.e. $[\![-]\!]$ turns a function in source to a unique constructor `Fun` of an inductively defined data type in target, and each free variable in $\lambda x : A.\ e$ corresponds to an argument to `Fun`.

Ideally, $[\![-]\!]$ should be type-preserving – well-typed source programs are translated to well-typed target programs. Clearly, the hypothesis of having a type-preserving transformation where $[\![\Pi x : A.\, B]\!] = \texttt{Pi}\ [\![A]\!]\ [\![B]\!]$ does not hold, since it translates functions of type $A \to (A \to A)$ to constructors returning $\texttt{Pi}\ [\![A]\!]\ (\texttt{Pi}\ [\![A]\!]\ [\![A]\!]\ )$, which fails the positivity check for inductive definitions. In fact, we cannot translate $\Pi x : A.\, B$ to an element of any inductive family in a type-preserving way. I proof this with the help of the following lemma.

Lemma 3.2.1 (Universe Preservation)**.** Type-preserving transformation $[\![-]\!]$ turns a universe in CC to a universe in Agda and it preserves the universe hierarchy.

Proof. We have $\cdot \vdash 0 : Nat : U_0 : U_1 : \cdots$ in CC (with freely-introduced natural numbers as a base type). To satisfy type preservation, we must have $[\![0]\!] : [\![Nat]\!] : [\![U_0]\!] : [\![U_1]\!] : \cdots$ in the target language. Therefore, $[\![U_0]\!], [\![U_1]\!], \cdots$ are universes (since they are the types of types) and they form a universe hierarchy in the target language. □

By universe preservation, for all CC expression e_1, e_2, if e_1 is in a higher universe than e_2:

$$\begin{aligned} &\Gamma \vdash e_1 : A, \quad \Gamma \vdash A : U_m, \\ &\Gamma \vdash e_2 : B, \quad \Gamma \vdash B : U_n\ (m > n), \end{aligned}$$

then $[\![e_1]\!]$ will also be in a higher universe than $[\![e_2]\!]$ in Agda.

$$\begin{aligned} &[\![e_1]\!] : [\![A]\!], \quad [\![A]\!] : \mathsf{Set_i}, \\ &[\![e_2]\!] : [\![B]\!], \quad [\![B]\!] : \mathsf{Set_j}\ (\mathsf{i} > \mathsf{j}). \end{aligned}$$

Now, I proof that type-preserving defunctionalization cannot be defined using inductive families.

Theorem 3.2.2. If $[\![-]\!]$ is type-preserving, then $[\![\Pi x : A.\, B]\!]$ cannot be an element of any inductive family.

Proof. Assume that $[\![-]\!]$ is type-preserving, $[\![\Pi x : A.\, B]\!]$ is an element of an inductive family `D`, and $\Gamma \vdash \Pi x : A.\, B : U_m$. Assume $[\![U_m]\!] = \mathtt{Set_n}$, which is reasonable by Lemma 3.2.1. Data type `D` must have the following general form.

```
data D : (y₁ : T₁) -> ··· -> (yₙ : Tₙ) -> Setₙ
```

`D` is in universe $\mathtt{Set_n}$ because $[\![-]\!]$ is type preserving. Pick an arbitrary function f in the source language and suppose that its type is $\Pi x : A.\, B$. By assumption, $[\![f]\!] = \mathtt{Fun}$, which is a valid constructor of `D`. The general form of `Fun` is

```
Fun : (z₁ : S₁) -> ··· -> (zₘ : Sₘ) -> (D t₁ ··· tₙ)
```

where each $(\mathtt{z_i}{:}\mathtt{S_i})$ corresponds to a free variable in f. Since `Fun` is a valid constructor, the universe of each $\mathtt{S_i}$ must be smaller than $\mathtt{Set_n}$ to pass the universe check. However, this is not true in general: f may contain free variables from universes higher than U_m, which are translated to terms in universes higher than $\mathtt{Set_n}$, by Lemma 3.2.1. Therefore, we have a contradiction. □

A type-preserving transformation cannot take $[\![\Pi x : A.\, B]\!] = \mathtt{Pi}\ [\![A]\!]\ [\![B]\!]$, since some results of this translation fail the positivity check. Theorem 3.2.2 shows that no inductive family can be used for type-preserving defunctionalization, and the key problem is the universe check that forbids impredicative constructors. If a function contains free variables in universes higher than the universe of the function's type, this function will be translated to a constructor that takes arguments in universes higher than the universe of its inductive family, which fails the universe check. Previous defunctionalization methods relied on the fact that any type can be an argument to a constructor, which is impredicative in nature.

Chapter 4

Abstract defunctionalization

This chapter presents my work on *abstract defunctionalization.* Abstract defunctionalization consists of the target language, *the Defunctionalized Calculus of Constructions (DCC)*, and a type-preserving defunctionalization transformation from the source language (CC) to DCC.

Section 4.1 introduces abstract defunctionalization and informally explains the design of DCC and the transformation. Section 4.2 presents the formal definition of DCC and Section 4.3 defines the defunctionalization transformation.

4.1 Main ideas

Abstractly speaking, defunctionalization is a transformation that eliminates functions by replacing them with a special kind of first-class value, which I refer to as *labels.* Each source-program function uniquely corresponds to a label in the target program. A label carries a list of values assigned to its corresponding function's free variables, similar to a function closure with the closure environment. The transformation turns applications of functions to arguments into applications of labels to (transformed) arguments, and it provides a mechanism for evaluating label applications.

Concretely, Pottier and Gauthier's polymorphic defunctionalization uses constructors of a generalized abstract data type (GADT) as labels, and the mechanism for evaluating label applications is an auxiliary *apply* function. Chapter 3 discussed an attempt of adapting this method to the dependently-typed world using inductive families (the counterpart of GADTs in dependently-typed languages) as labels. This turned out to be a failure, because constructors of inductive families cannot accept arguments from universes higher than the universe of the inductive definition, but functions could contain free variables from arbitrarily high universes.

One alternative approach is to design a target language with a primitive notion of labels that fits the abstract description of defunctionalization. Defining abstract transformations with specialized target languages is a common approach in the literature. For example, Minamide et al. presented a type-preserving closure conversion from the simply-typed lambda calculus to a typed intermediate language with closures as first-class values [26]. They named this method *abstract closure conversion*, and Bowman and Ahmed adapted it to dependently-typed closure conversion [11].

Abstract defunctionalization takes inspiration from works of Minamide et al. and Bowman and Ahmed. It consists of a target language, the Defunctionalized Calculus of Constructions (DCC), and a type-preserving transformation from CC into it. Before formally presenting DCC's syntax and type judgements, I give an informal explaination of its key features and the intuition behind them. I write DCC expressions in a san-serif blue font to distinguish them from the source language expressions.

DCC is a language similar to CC, except that its syntax contains first-class function labels instead of lambda abstractions. The syntax of a label $\mathfrak{L}\{\mathsf{e}_1, \cdots, \mathsf{e}_n\}$ consists of a label name ($\mathfrak{L}$) and a list of free-variable terms it carries ($\{\mathsf{e}_1, \cdots, \mathsf{e}_n\}$). DCC also provides label contexts as the mechanism for evaluating label applications. These contexts contain elements in the following form.

$$\mathfrak{L} \mapsto (\{\mathsf{x}_1 : \mathsf{A}_1, \cdots, \mathsf{x}_n : \mathsf{A}_n\}, \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}), \mathsf{e})$$

If $\mathfrak{L}$ corresponds to a function f in the source language, then $\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\}$, $\mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})$, and e correspond to the types of free variables in f, f's type, and f's body respectively. Applying a label to an argument evaluates to e with all the free-variable terms substituted in.

For example, the label context associated with the defunctionalized natural-number addition function $\lambda x : Nat.\ (\lambda y : Nat.\ x + y)$ is the following.

$$\begin{aligned}
&\mathfrak{L}_1 \mapsto (\{\mathsf{x} : \mathsf{Nat}\}, \mathsf{Pi}(\mathsf{y},\mathsf{Nat},\mathsf{Nat}), \mathsf{x} + \mathsf{y}),\\
&\mathfrak{L}_0 \mapsto (\{\}, \mathsf{Pi}(\mathsf{x}, \mathsf{Nat}, \mathsf{Pi}(\mathsf{y},\mathsf{Nat},\mathsf{Nat})), \mathfrak{L}_1\{\mathsf{x}\})
\end{aligned}$$

Informally, this label context says that $\mathfrak{L}_0$ corresponds to a function that has no free variables and has the type $\Pi x : Nat.\, \Pi y : Nat.\, Nat$. Given an input x, it returns another function whose free variable is assigned with the value of x. The interpretation for $\mathfrak{L}_1$ is similar.

The reader might find the resemblance between DCC and defunctionalized programs in OCaml. Intuitively, a label term $\mathfrak{L}\{\mathsf{e}_1, \cdots, \mathsf{e}_n\}$ is just like a constructor and all the arguments it takes (`Fun e1` $\cdots$ `en`), and the label context is like the pattern-matching cases of the `apply` function. The difference is that a function's type, the types of its free variables, and the function body are encoded implictly in the type of `Fun` and the definition of `apply`, whereas DCC places them explicitly in the label context.

4.2 Defunctionalized Calculus of Constructions (DCC)

This section presents the formal definition of the Defunctionalized Calculus of Constructions.

4.2.1 Syntax

DCC is a language similar to CC, except that its syntax (Figure 4.1) contains first-class function labels instead of lambda abstractions. DCC has an infinite hierarchy of predicative universes U_i, and its term expressions are: variables x (from an infinite set of variable names), universes U, Π-types $\mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})$, applications $\mathsf{Apply}\ \mathsf{e}_1\ \mathsf{e}_2$, and labels $\mathfrak{L}\{\bar{\mathsf{e}}\}$. I omit base types for simplicity. Except for labels, all term expressions in DCC are the same as their counterparts in CC (only written down in a different way).

A label expression $\mathfrak{L}\{\bar{\mathsf{e}}\}$ is a label name $\mathfrak{L}$ supplied with a list of terms $\mathsf{e}_1, \cdots, \mathsf{e}_n$ assigned to its free variables. Labels can take zero free variables, but a label name by itself is not a term. In other words, $\mathfrak{L}$ is not a term, but $\mathfrak{L}\{\}$ is. I write $\bar{\mathsf{e}}$ as an abbreviation of a list of zero or more expressions. Label names $\mathfrak{L}_1, \mathfrak{L}_2, \cdots$ come from an infinite set of names which is disjoint to the set of variable names, and I write labels in a different font from variables to emphasise this.

A DCC context $\mathfrak{D}\,;\Gamma$ consists of a context Γ for types and a context $\mathfrak{D}$ for label definitions. The type context is a list of variable-expression pairs. The label context is a list of label names $\mathfrak{L}$ and their associated data $(\{\bar{\mathsf{x}}:\bar{\mathsf{A}}\}, \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}), \mathsf{e})$. The first part $\{\bar{\mathsf{x}}:\bar{\mathsf{A}}\}$ records the types of free variables the label takes. The second part $\mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})$ specifies the type of the label, and the third part e is what the label reduces to when it is applied to an argument.

I use $\bar{\mathsf{x}}:\bar{\mathsf{A}}$ as a syntactic sugar for $\mathsf{x}_1:\mathsf{A}_1, \cdots, \mathsf{x}_n:\mathsf{A}_n$. Note that A_{i+1} could depend on $\mathsf{x}_1, \cdots, \mathsf{x}_i$, and the list of free variables can be empty. Variables $\bar{\mathsf{x}}$ are bound to A, B, and e; the variable x in the second part $\mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})$ is bound to *both* B and e.

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

$$\mathfrak{L}\{\bar{\mathsf{e}}\}[\mathsf{e}/\mathsf{x}] \triangleq \mathfrak{L}\{\bar{\mathsf{e}}[\mathsf{e}/\mathsf{x}]\},$$

where $\bar{\mathsf{e}}[\mathsf{e}/\mathsf{x}]$ is a syntactic sugar for $\mathsf{e}_1[\mathsf{e}/\mathsf{x}], \cdots, \mathsf{e}_n[\mathsf{e}/\mathsf{x}]$.

$$
\begin{array}{llcl}
\textit{Universes} & \mathsf{U} & ::= & \mathsf{U_i} \\
\textit{Expressions} & \mathsf{e},\ \mathsf{A},\ \mathsf{B} & ::= & \mathsf{x} \mid \mathsf{U} \mid \mathsf{Pi(x,A,B)} \mid \mathsf{Apply}\ \mathsf{e_1}\ \mathsf{e_2} \mid \mathfrak{L}\{\bar{\mathsf{e}}\} \\
\textit{Type contexts} & \Gamma & ::= & \cdot \mid \Gamma,\ \mathsf{x}:\mathsf{A} \\
\textit{Label contexts} & \mathfrak{D} & ::= & \cdot \mid \mathfrak{D},\ \mathfrak{L} \mapsto (\{\bar{\mathsf{x}}:\bar{\mathsf{A}}\},\ \mathsf{Pi(x,A,B)},\ \mathsf{e}) \\
\textit{DCC contexts} & \mathfrak{D}\,;\Gamma & &
\end{array}
$$

Figure 4.1: DCC syntax

4.2.2 Type judgements

DCC's type judgements are of the form $\mathfrak{D}\,;\Gamma \vdash \mathsf{e}:\mathsf{A}$, and typing rules are given in Figure 4.2. Rules for variables, universes, Π-types, applications, and conversion are identical to their counterpart rules in CC, so I focus on the rule for labels. A label term $\mathfrak{L}\{\bar{\mathsf{e}}\}$ is well-typed in $\mathfrak{D}\,;\Gamma$ the following conditions are satisfied.

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

Specifically, condition (4) means:

$$
\begin{aligned}
&\mathfrak{D}\,;\Gamma \vdash \mathsf{e_1}:\mathsf{A_1}, \\
&\mathfrak{D}\,;\Gamma \vdash \mathsf{e_2}:\mathsf{A_2}[\mathsf{e_1}/\mathsf{x_1}], \\
&\qquad \cdots \\
&\mathfrak{D}\,;\Gamma \vdash \mathsf{e_n}:\mathsf{A_n}[\mathsf{e_1}/\mathsf{x_1},\cdots,\mathsf{e_{n-1}}/\mathsf{x_{n-1}}].
\end{aligned}
$$

Each $\mathsf{A_{i+1}}$ depends on $\mathsf{x_1},\cdots,\mathsf{x_i}$, so $\mathsf{e_1},\cdots,\mathsf{e_i}$ need to be substituted in $\mathsf{A_{i+1}}$ in the type judgement for $\mathsf{e_{i+1}}$. The type of $\mathfrak{D}\,;\Gamma \vdash \mathsf{e}:\mathsf{A}$ is

$$\mathsf{Pi(x,A[\bar{e}/\bar{x}],B[\bar{e}/\bar{x}])}.$$

Note that values of free variables $\bar{\mathsf{e}}$ are substituted in $\mathsf{Pi(x,A,B)}$, the specified type of the label. I use $[\bar{\mathsf{e}}/\bar{\mathsf{x}}]$ as a syntactic sugar of $[\mathsf{e_1}/\mathsf{x_1},\ \cdots,\ \mathsf{e_n}/\mathsf{x_n}]$, and conditions (3) and (4) are abbreviated to $\mathfrak{D}\,;\Gamma \vdash \bar{\mathsf{e}}:\bar{\mathsf{A}}$ as a convention.

The DCC judgement for well-formed contexts is $\vdash \mathfrak{D}\,;\Gamma$ and the judgement for well-formed label contexts is $\vdash \mathfrak{D}$. Their derivation rules are given in Figure 4.3. A well-formed label

$$\frac{\mathsf{x}:\mathsf{A} \in \Gamma \qquad \vdash \mathfrak{D}\,;\Gamma}{\mathfrak{D}\,;\Gamma \vdash \mathsf{x}:\mathsf{A}} \quad (\mathsf{Var})$$

$$\frac{\vdash \mathfrak{D}\,;\Gamma}{\mathfrak{D}\,;\Gamma \vdash \mathsf{U_i}:\mathsf{U_{i+1}}} \quad (\mathsf{Universe})$$

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

$$\frac{\mathfrak{D}\,;\Gamma \vdash \mathsf{e_1} : \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}) \qquad \mathfrak{D}\,;\Gamma \vdash \mathsf{e_2}:\mathsf{A}}{\mathfrak{D}\,;\Gamma \vdash \mathsf{Apply}\ \mathsf{e_1}\ \mathsf{e_2} : \mathsf{B}[\mathsf{e_2}/\mathsf{x}]} \quad (\mathsf{Apply})$$

$$\frac{\begin{array}{cc}\vdash \mathfrak{D}\,;\Gamma & \mathfrak{D}\,;\Gamma \vdash \bar{\mathsf{e}}:\bar{\mathsf{A}} \\ \multicolumn{2}{c}{\mathfrak{L} \mapsto (\{\bar{\mathsf{x}}:\bar{\mathsf{A}}\},\ \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}),\ \mathsf{e}) \in \mathfrak{D}}\end{array}}{\mathfrak{D}\,;\Gamma \vdash \mathfrak{L}\{\bar{\mathsf{e}}\} : \mathsf{Pi}(\mathsf{x},\mathsf{A}[\bar{\mathsf{e}}/\bar{\mathsf{x}}],\mathsf{B}[\bar{\mathsf{e}}/\bar{\mathsf{x}}])} \quad (\mathsf{Label})$$

$$\frac{\mathfrak{D}\,;\Gamma \vdash \mathsf{e}:\mathsf{A} \qquad \mathfrak{D}\,;\Gamma \vdash \mathsf{B}:\mathsf{U} \qquad \mathfrak{D}\,;\Gamma \vdash \mathsf{A} \equiv \mathsf{B}}{\mathfrak{D}\,;\Gamma \vdash \mathsf{e}:\mathsf{B}} \quad (\mathsf{Equiv})$$

Figure 4.2: DCC Typing

$$\frac{\vdash \mathfrak{D}}{\vdash \mathfrak{D}\,;\cdot}\ (\mathsf{WF\text{-}Empty}) \qquad \frac{\mathfrak{D}\,;\Gamma \vdash \mathsf{A}:\mathsf{U}}{\vdash \mathfrak{D}\,;\Gamma,\ \mathsf{x}:\mathsf{A}}\ (\mathsf{WF\text{-}Type}) \qquad \frac{}{\vdash \cdot}\ (\mathsf{WFL\text{-}Empty})$$

$$\frac{\mathfrak{D}\,;\Gamma_{\mathsf{fv}} \vdash \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}) : \mathsf{U} \qquad \mathfrak{D}\,;\Gamma_{\mathsf{fv}},\ \mathsf{x}:\mathsf{A} \vdash \mathsf{e}:\mathsf{B}}{\vdash \mathfrak{D},\ \mathfrak{L} \mapsto (\{\bar{\mathsf{x}}:\bar{\mathsf{A}}\},\ \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}),\ \mathsf{e})}\ (\mathsf{WFL\text{-}Label})$$
$$\text{where}\ \ \Gamma_{\mathsf{fv}} \triangleq \cdot,\ \bar{\mathsf{x}}:\bar{\mathsf{A}}$$

Figure 4.3: Well-formedness of DCC contexts and label contexts

context with an empty type context is a well-formed DCC context (WF-Empty). If $\mathfrak{D}\,;\Gamma$ is well-formed and A is a valid type in it, then $\mathfrak{D}\,;\Gamma,\ \mathsf{x}:\mathsf{A}$ is also well-formed (WF-Type).

For label contexts, the empty context is well-formed (WFL-Empty). A label context extended with a new entry $\mathfrak{L} \mapsto (\{\bar{\mathsf{x}}:\bar{\mathsf{A}}\},\ \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}),\ \mathsf{e})$ is well-formed if $\mathfrak{L}$ is not present in $\mathfrak{D}$, $\mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})$ is a valid type in $\mathfrak{D}\,;\Gamma_{\mathsf{fv}}$, and the type of e is B in $\mathfrak{D}\,;\Gamma_{\mathsf{fv}},\ \mathsf{x}:\mathsf{A}$ (WFL-Label). Here, Γ_{fv} is the *free-variable type context* formed by the label's free variables, defined as $\Gamma_{\mathsf{fv}} = \cdot,\ \bar{\mathsf{x}}:\bar{\mathsf{A}}$. Note that the well-formedness rules implicitly specify that expressions in Γ may refer to labels in $\mathfrak{D}$, but expressions in $\mathfrak{D}$ cannot refer to variables in Γ.

Both the type context and the label context have the weakening property – a well-typed term is still well-typed in an extended type or label context.

Lemma 4.2.1 (Type weakening)**.** If $\mathfrak{D}\,;\Gamma \vdash \mathsf{e}:\mathsf{A}$ and $\mathfrak{D}\,;\Gamma \vdash \mathsf{B}:\mathsf{U}$, then $\mathfrak{D}\,;\Gamma,\ \mathsf{x}:\mathsf{B} \vdash \mathsf{e}:\mathsf{A}$.

Lemma 4.2.2 (Label weakening)**.** If $\mathfrak{D}\,;\Gamma \vdash \mathsf{e}:\mathsf{C}$, $\mathfrak{D}\,;\Gamma_{\mathsf{fv}} \vdash \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}) : \mathsf{U}$, and $\mathfrak{D}\,;\Gamma_{\mathsf{fv}},\ \mathsf{x}:\mathsf{A} \vdash \mathsf{e}':\mathsf{B}$, then $(\mathfrak{D},\ \mathfrak{L} \mapsto (\{\bar{\mathsf{x}}:\bar{\mathsf{A}}\},\ \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}),\ \mathsf{e}'))\,;\Gamma \vdash \mathsf{e}:\mathsf{C}$.

$$\mathfrak{D};\Gamma \vdash \mathsf{Apply}\ \mathfrak{L}\{\bar{e}\}\ e_2 \ \triangleright\ e_1[\bar{e}/\bar{x},\ e_2/x] \qquad \text{(Beta)}$$
$$\text{where } \mathfrak{L} \mapsto (\{\bar{x}:\bar{A}\},\ \mathsf{Pi}(x,A,B),\ e_1) \in \mathfrak{D}$$

$$\frac{\mathfrak{D};\Gamma \vdash e_1 \triangleright^* e \qquad \mathfrak{D};\Gamma \vdash e_2 \triangleright^* e}{\mathfrak{D};\Gamma \vdash e_1 \equiv e_2} \qquad \text{(Eq-reduce)}$$

$$\frac{\begin{array}{l}\mathfrak{D};\Gamma \vdash e_1 \triangleright^* \mathfrak{L}\{\bar{e}\} \qquad \mathfrak{D};\Gamma \vdash e_2 \triangleright^* e_2' \\ \mathfrak{L} \mapsto (\{\bar{x}:\bar{A}\},\ \mathsf{Pi}(x,A,B),\ e) \in \mathfrak{D} \\ \mathfrak{D};\Gamma,\ x:A[\bar{e}/\bar{x}] \vdash e[\bar{e}/\bar{x}] \equiv \mathsf{Apply}\ e_2'\ x\end{array}}{\mathfrak{D};\Gamma \vdash e_1 \equiv e_2} \qquad \text{(Eq-Eta1)}$$

$$\frac{\begin{array}{l}\mathfrak{D};\Gamma \vdash e_1 \triangleright^* e_1' \qquad \mathfrak{D};\Gamma \vdash e_2 \triangleright^* \mathfrak{L}\{\bar{e}\} \\ \mathfrak{L} \mapsto (\{\bar{x}:\bar{A}\},\ \mathsf{Pi}(x,A,B),\ e) \in \mathfrak{D} \\ \mathfrak{D};\Gamma,\ x:A[\bar{e}/\bar{x}] \vdash \mathsf{Apply}\ e_1'\ x \equiv e[\bar{e}/\bar{x}]\end{array}}{\mathfrak{D};\Gamma \vdash e_1 \equiv e_2} \qquad \text{(Eta2)}$$

Figure 4.4: DCC reduction and equivalence

DCC's reductions $\mathfrak{D};\Gamma \vdash e_1 \triangleright e_2$ and equivalence $\mathfrak{D};\Gamma \vdash e_1 \equiv e_2$ are defined in Figure 4.4. There is only one reduction rule in DCC: $\mathsf{Apply}\ \mathfrak{L}\{\bar{e}\}\ e_2$ reduces to $e_1[\bar{e}/\bar{x},\ e_2/x]$, where e_1 is found in the label's associated data $\mathfrak{L} \mapsto (\{\bar{x}:\bar{A}\},\ \mathsf{Pi}(x,A,B),\ e_1)$ in the label context $\mathfrak{D}$. A reduction sequence is noted as $e_1 \triangleright^* e_2$, which means e_1 reduces to e_2 in zero or more steps.

Two terms e_1 and e_2 are equivalent if they both reduce to the same term e in a reduction sequence or they are η-equivalent. DCC's η-equivalence rules are similar to that of CC. Rule (Eq-Eta1) defines that e_1 and e_2 are equivalent if they satisfy the following conditions.

1. e_1 reduces to a label $\mathfrak{L}\{\bar{e}\}$ in a reduction sequence.
2. e_2 reduces to some expression e_2' in a reduction sequence.
3. $\mathfrak{L}$ associates with $(\{\bar{x}:\bar{A}\},\ \mathsf{Pi}(x,A,B),\ e)$ in the label context $\mathfrak{D}$.
4. $\mathsf{Apply}\ e_2'\ x$ is equivalent to $e[\bar{e}/\bar{x}]$ in $\mathfrak{D};\Gamma,\ x:A$.

Rules (Eq-Eta1) and (Eq-Eta2) are symmetrical.

4.3 Transformation

This section gives the definition of dependently typed defunctionalization transformation. The transformation consists of two parts: a transformation $[\![-]\!]$ for terms and a meta-function $[\![-]\!]_d$ that extracts function definitions from the source program. The

$$\frac{}{\Gamma \vdash x : A \rightsquigarrow \mathsf{x}} \quad \text{(T-Var)}$$

$$\frac{}{\Gamma \vdash U_i : U_{i+1} \rightsquigarrow \mathsf{U_i}} \quad \text{(T-Universe)}$$

$$\frac{\Gamma \vdash A : U_i \rightsquigarrow \mathsf{A} \qquad \Gamma, x : A \vdash B : U_j \rightsquigarrow \mathsf{B}}{\Gamma \vdash \Pi x : A.\ B : U_{max(i,j)} \rightsquigarrow \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})} \quad \text{(T-Pi)}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : A.\ B \rightsquigarrow \mathsf{e_1} \qquad \Gamma \vdash e_2 : A \rightsquigarrow \mathsf{e_2}}{\Gamma \vdash e_1\ e_2 : B[e2/x] \rightsquigarrow \mathsf{Apply}\ \mathsf{e_1}\ \mathsf{e_2}} \quad \text{(T-Apply)}$$

$$\frac{\mathrm{FV}(\lambda^i x : A.\ e) = \bar{x} : \bar{A} \qquad \Gamma \vdash \bar{x} : \bar{A} \rightsquigarrow \bar{\mathsf{x}}}{\Gamma \vdash \lambda^i x : A.\ e : \Pi x : A.\ B \rightsquigarrow \mathfrak{L}_{\mathsf{i}}\{\bar{\mathsf{x}}\}} \quad \text{(T-Lambda)}$$

$$\frac{\Gamma \vdash e : A \rightsquigarrow \mathsf{e}}{\Gamma \vdash e : B \rightsquigarrow \mathsf{e}} \quad \text{(T-Equiv)}$$

Figure 4.5: Defunctionalization transformation

term transformation produces the target program and the meta-function $[\![-]\!]_d$ gives a label context, which functions similarly to the auxiliary function *apply* in polymorphic defunctionalization.

4.3.1 Term transformation

The term transformation $[\![-]\!]$ turns expressions in CC into expressions in DCC. I define the transformation with a new judgement of the form $\Gamma \vdash e : A \rightsquigarrow \mathsf{e}$, and $[\![e]\!] \triangleq \mathsf{e}$. Figure 4.5 gives the derivation rules of this judgement.

The term 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. I use a convention that every function in the transformation's input e is tagged with a unique identifier i ($i \in \mathbb{N}$), and its corresponding label's name is $\mathfrak{L}_{\mathsf{i}}$ (α-equivalent functions get the same tag).

The transformation turns a function $(\lambda^i x : A.\ e)$ into a label $\mathfrak{L}_{\mathsf{i}}\{\bar{\mathsf{x}}\}$, where $\bar{\mathsf{x}}$ come from the function's free variables $\bar{x}$ (T-Lambda). The meta-function FV (see Definition 4.3.1) 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, FV(e) must recursively work out all the variables needed for e to be well-typed.

Definition 4.3.1. FV(e) takes $\Gamma \vdash e : A$, the type judgement of e, as an implicit argument. It firstly computes all the unbound variables $x_1, \cdots, x_n$ in e 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.

$$\begin{aligned}
\mathrm{FV}(e) \quad = \quad & \mathrm{FV}(A_1) \cup \cdots \cup \mathrm{FV}(A_n) \cup \Gamma_{fv} \\
\text{where} \quad & \mathit{fv}\ (e) \cup \mathit{fv}\ (A) = x_1, \cdots, x_n \\
& \Gamma \vdash x_1 : A_1, \cdots, \Gamma \vdash x_n : A_n \\
& \Gamma_{fv} \triangleq x_1 : A_1, \cdots, x_n : A_n.
\end{aligned}$$

Here, the union of two type contexts $\Gamma_1 \cup \Gamma_2$ is Γ_1 appended with all the variable-expression pairs $x : A$ that only appear in Γ_2 in the order of their apperence. Variable x would always be paired with the same expression A in Γ_1 and Γ_2 in this usage, because all pairs $x : A$ in Γ_1 and Γ_2 come from the same context Γ. Since FV(e) gives all the variables needed to correctly type e, $\Gamma \vdash e : A$ implies that $\mathrm{FV}(e) \vdash e : A$.

Lemma 4.3.2. If $\Gamma \vdash e : A$, then $\mathrm{FV}(e) \vdash e : A$.

4.3.2 Extracting function definitions

Defunctionalization is not complete with just the term transformation which turns functions into labels but throws away the function body. The meta-function $[\![-]\!]_d$ takes a CC-expression and returns a label context $\mathfrak{D}$. For every function $(\lambda^i x : A.\ e)$ in the source program, $\mathfrak{D}$ contains the implementation of that function as an item in the following form, where $\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\}$, $\mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})$, and e correspond to the free variables $(\bar{x} : \bar{A})$ in λ^i, the type of λ^i and the function body e respectively.

$$\mathfrak{L}_\mathsf{i} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\}, \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}), \mathsf{e})$$

Function extraction is more subtle than transforming terms. Functions may appear in the type of an expression, even if the expression itself does not contain that function! For example, consider the following CC context and expression (with base type natural numbers Nat).

$$\begin{aligned}
\Gamma \ &\triangleq\ \cdot,\ A : (Nat \to Nat) \to U_0,\ a : \Pi f : (Nat \to Nat).\, A\ (\lambda n : Nat.\ (f\ n) + 1) \\
e \ &\triangleq\ a\ (\lambda x : Nat.\ x + 1)
\end{aligned}$$

$$\frac{\Gamma \vdash A : U \leadsto_d \mathfrak{D}}{\Gamma \vdash x : A \leadsto_d \mathfrak{D}} \quad \text{(D-Var)}$$

$$\frac{}{\Gamma \vdash U_i : U_{i+1} \leadsto_d \cdot} \quad \text{(D-Universe)}$$

$$\frac{\Gamma \vdash A : U_i \leadsto_d \mathfrak{D}_\mathsf{A} \qquad \Gamma, x : A \vdash B : U_j \leadsto_d \mathfrak{D}_\mathsf{B}}{\Gamma \vdash \Pi x : A.\ B : U_{max(i,j)} \leadsto_d \mathfrak{D}_\mathsf{A} \cup \mathfrak{D}_\mathsf{B}} \quad \text{(D-Pi)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \Pi x : A.\ B \leadsto_d \mathfrak{D}_1 \qquad \Gamma \vdash e_2 : A \leadsto_d \mathfrak{D}_2 \\ \Gamma \vdash B[e2/x] : U \leadsto_d \mathfrak{D}_3\end{array}}{\Gamma \vdash e_1\ e_2 : B[e2/x] \leadsto_d \mathfrak{D}_1 \cup \mathfrak{D}_2 \cup \mathfrak{D}_3} \quad \text{(D-Apply)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash A : U \leadsto_d \mathfrak{D}_\mathsf{A} & \Gamma, x : A \vdash e : B \leadsto_d \mathfrak{D}_\mathsf{e} \\ \mathrm{FV}(\lambda^i x : A.\ e) = \bar{x} : \bar{A} & \Gamma \vdash \bar{x} : \bar{A} \leadsto \bar{\mathsf{x}} : \bar{\mathsf{A}} \\ \Gamma \vdash \Pi x : A.\ B \leadsto \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}) & \Gamma, x : A \vdash e \leadsto \mathsf{e}\end{array}}{\begin{array}{c}\Gamma \vdash \lambda^i x : A.\ e : \Pi x : A.\ B \leadsto_d \\ \mathfrak{D}_\mathsf{A} \cup \mathfrak{D}_\mathsf{e},\ \mathfrak{L}_\mathsf{i} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\},\ \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}),\ \mathsf{e})\end{array}} \quad \text{(D-Lambda)}$$

$$\frac{\Gamma \vdash e : A \leadsto_d \mathfrak{D} \qquad \Gamma \vdash B : U \leadsto_d \mathfrak{D}_\mathsf{B}}{\Gamma \vdash e : B \leadsto_d \mathfrak{D} \cup \mathfrak{D}_\mathsf{B}} \quad \text{(D-Equiv)}$$

Figure 4.6: Extracting function definitions

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

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

Hence, a new function definition appeared in the type of e as the result of a substitution! This new function should be included in the label context $\mathfrak{D}$ to achieve type preservation. Extracting function definitions in e involes finding every function that appeared in the type derivation of e. In other words, the transformation defunctionalizes not just the source-language expression, but its entire type derivation tree.

I define $[\![-]\!]_d$ with a new judgement of the form $\Gamma \vdash e : A \leadsto_d \mathfrak{D}$, and $[\![e]\!]_d \triangleq \mathfrak{D}$. Figure 4.6 gives the derivation rules of this judgement.

Variables do not contain function definitions, so the definitions in x are just the definitions in its type A (D-Var). Type derivations of universes do not involve functions at all (D-

Universe). 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 4.3.1), and there is no ambiguity since different functions correspond to different label names.

Definitions in an application $e_1\ e_2$ are the union of definitions in e_1, e_2, and $B[e_2/x]$, since substitution creates new function definitions (D-Apply). For a lambda abstraction $\lambda^i x : A.\ e$, the definitions it contains are the union of definitions in e and A appended with $\mathfrak{L}_\mathsf{i}$, the definition of itself (D-Lambda). If e has type B by the conversion rule, then the definitions involved in the derivation of $\Gamma \vdash e : B$ are the union of definitions in the derivation of $\Gamma \vdash e : A$ and definitions in B (D-Equiv).

I define the subset relation of label contexts, which helps to state definitions and theorems in the remainder of this dissertation.

Definition 4.3.3. For two well-formed label contexts $\mathfrak{D}_1$ and $\mathfrak{D}_2$, $\mathfrak{D}_1 \subseteq \mathfrak{D}_2$ if for all $\mathfrak{L}_\mathsf{i} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\}, \mathsf{Pi}(\mathsf{x}, \mathsf{A}, \mathsf{B}), \mathsf{e})$ in $\mathfrak{D}_1$, $\mathfrak{L}_\mathsf{i} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\}, \mathsf{Pi}(\mathsf{x}, \mathsf{A}, \mathsf{B}), \mathsf{e})$ is also in $\mathfrak{D}_2$.

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

Lemma 4.3.4 (Label context weakening (subsets))**.** If $\mathfrak{D}_1 ; \Gamma \vdash \mathsf{e} : \mathsf{A}$, $\vdash \mathfrak{D}_2$, and $\mathfrak{D}_1 \subseteq \mathfrak{D}_2$, then $\mathfrak{D}_2 ; \Gamma \vdash \mathsf{e} : \mathsf{A}$.

Since the transformation defunctionalizes the entire type derivation tree of a term, if $\Gamma \vdash e : A$, then all elements in $[\![A]\!]_d$ are also in $[\![e]\!]_d$. This property requires a proof as it is not immediately obvious from the definition that this is true for case (D-Lambda).

Lemma 4.3.5. For all well-typed terms $\Gamma \vdash e : A$ in CC, $[\![A]\!]_d \subseteq [\![e]\!]_d$.

Proof. By induction on rules defined in Figure 4.6. All cases except (D-Lambda) are either trivially true or follows simply from the definition.

The goal in case (D-Lambda) is to show that $[\![\Pi x : A.\, B]\!]_d \subseteq [\![\lambda^i x : A.\ e]\!]_d$. By assumption,

$$\Gamma \vdash \lambda^i x : A.\ e : \Pi x : A.\, B \ \rightsquigarrow_d \ \mathfrak{D}_\mathsf{A} \cup \mathfrak{D}_\mathsf{e},\ \mathfrak{L}_\mathsf{i} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\}, \mathsf{Pi}(\mathsf{x}, \mathsf{A}, \mathsf{B}), \mathsf{e}).$$

In other words, $[\![\lambda^i x : A.\ e]\!]_d = [\![A]\!]_d \cup [\![e]\!]_d$ with definition of λ^i appended to it. By definition, $[\![\Pi x : A.\, B]\!]_d = [\![A]\!]_d \cup [\![B]\!]_d$, and hence $[\![A]\!]_d \subseteq [\![\lambda^i x : A.\ e]\!]_d$. We have $[\![B]\!]_d \subseteq [\![e]\!]_d$ by the induction hypothesis, since $\Gamma, x{:}A \vdash e : B$. Therefore, $[\![B]\!]_d \subseteq [\![\lambda^i x : A.\ e]\!]_d$. □

The term transformation and the process of extracting function definitions ($[\![-]\!]$ and $[\![-]\!]_d$)

act pointwise on CC contexts. In other words,

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

Finally, I give an example of dependently typed defunctionalization to illustrate that the transformation is type-preserving and correct.

Example 4.3.1. Let the source program p be the an application of the polymorphic identity function in CC (see Example 2.2.1), and the lambda abstrctions are tagged with natural numbers.

$$\begin{aligned} & p \triangleq (\lambda^0 A : U_0.\ (\lambda^1 x : A.\ x))\ \mathit{Nat}\ 1 \\ & \cdot \vdash p : \mathit{Nat} \\ & \cdot \vdash p\ \triangleright (\lambda x : \mathit{Nat}.\ x)\ 1\ \triangleright\ 1 \end{aligned}$$

The source program is a function with no free variables applied to the base type Nat and then to 1. So, the term transformation is a label supplied with no free-variable terms applied to Nat and then to $\mathsf{1}$.

$$[\![p]\!] = \mathsf{Apply}\ (\mathsf{Apply}\ \mathfrak{L}_0\{\}\ \mathsf{Nat})\ \mathsf{1}$$

There are two function definitions (λ^0 and λ^1) in the derivation tree of the source program. Function λ^1 has one free variable A of type U_0; its type is $A \to A$ and its function body is x. Function λ^0 has no free variable; its type is $\Pi A : U_0.\ A \to A$ and its function body is λ^1 where its free variable A is assigned with the value of the input of λ^0. Therefore, $[\![p]\!]_d$ returns the following label context with two items.

$$\begin{aligned} \mathfrak{D} = \cdot\ ,\ \ & \mathfrak{L}_1 \mapsto (\{\mathsf{A} : \mathsf{U}_0\}, \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{A}), \mathsf{x}), \\ & \mathfrak{L}_0 \mapsto (\{\}, \mathsf{Pi}(\mathsf{A}, \mathsf{U}_0, \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{A})), \mathfrak{L}_1\{\mathsf{A}\}) \end{aligned}$$

If the transformation is type preserving and correct, the type of $[\![p]\!]$ should be Nat and it should reduce to $\mathsf{1}$ (in context $\mathfrak{D}\,;\cdot$). By (Apply), the type of ($\mathsf{Apply}\ \mathfrak{L}_0\{\}\ \mathsf{Nat}$) is $(\mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{A}))[\mathsf{Nat}/\mathsf{A}] = \mathsf{Pi}(\mathsf{x},\mathsf{Nat},\mathsf{Nat})$. So, the type of $[\![p]\!]$ is Nat. Also,

$$\mathfrak{D}\,;\cdot \vdash \mathsf{Apply}\ (\mathsf{Apply}\ \mathfrak{L}_0\{\}\ \mathsf{Nat})\ \mathsf{1}\ \triangleright\ \mathsf{Apply}\ \mathfrak{L}_1\{\mathsf{Nat}\}\ \mathsf{1}\ \triangleright\ \mathsf{1}.$$

Chapter 5

Type preservation, correctness, and consistency

In this chapter, I show that abstract defunctionalization is type preserving and correct (Section 5.1), and the target language DCC is type-safe and consistent (Section 5.2). For simplicity, I use $\mathsf{\Gamma}, \mathsf{e}$ to stand for $[\![\Gamma]\!], [\![e]\!]$ and $\mathfrak{D}_{\mathsf{\Gamma}}, \mathfrak{D}_{\mathsf{e}}$ to stand for $[\![\Gamma]\!]_d, [\![e]\!]_d$ when there is no ambiguity.

5.1 Correctness and type preservation

As shown in Example 4.3.1, it is easy to verify correctness and type preservation for a particular example program, but it is more difficult to prove that they hold for the transformation itself. Firstly, I give the definition of the correctness of the transformation.

Definition 5.1.1. Dependently typed defunctionalization is *correct* if for all base types A and values v of type A,

$$\cdot \vdash e : A \;\wedge\; \cdot \vdash e \;\triangleright^* v \implies (\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}}); \cdot \vdash \mathsf{e} \;\triangleright^* \mathsf{v}' \text{ where } \mathsf{v}' \equiv \mathsf{v}.$$

In other words, if a closed program e evaluates to a base-type value v, then e evaluates to a base-type value v' that is equivalent to v. Ideally, this follows as a corollary of the *preservation of reduction sequences.*

Definition 5.1.2. Dependently typed defunctionalization *preserves reduction sequences* if

$$\begin{aligned} \Gamma \vdash e_1 \;\triangleright^* e_2 \implies & (\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e_1}} \cup \mathfrak{D}_{\mathsf{e_2}}); \mathsf{\Gamma} \vdash \mathsf{e_1} \;\triangleright^* \mathsf{e}, \\ & \text{where } (\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e_1}} \cup \mathfrak{D}_{\mathsf{e_2}}); \mathsf{\Gamma} \vdash \mathsf{e} \equiv \mathsf{e_2}. \end{aligned}$$

Ideally, I could show this property by showing that the transformation preserves all small-step reductions $\Gamma \vdash e_1 \triangleright e_2$, followed by an induction on the number of reduction steps in a sequence. However, CC's meta-language substitution $(\lambda x : A.\ e_1)[e_2/y]$ creates a new function definition when it substitutes an expression into a free variable of a function. So, for a reduction sequence $e_1 \triangleright \cdots \triangleright e_n$ in CC, some e_i may contain function definitions that are not exist in e_1 or e_n. This means that not all e_i translate into well-typed DCC terms e_i in $(\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}_1} \cup \mathfrak{D}_{\mathsf{e}_\mathsf{n}});\mathsf{\Gamma}$, which makes proofs by induction difficult. 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 source of creating new function definitions in CC's reduction sequences. There would be no problem if the source language does not evaluate substitutions into functions but keeps them as primitive expressions instead. To use this observation formally, I define a helper language CC_S, which is an extension of CC with explicit substitutions [27]. In addition, this language does not reduce substitutions of expressions into functions.

Since CC_S extends CC, every CC expression is trivially a CC_S expression. I denote this trivial transformation from CC to CC_S as σ. Then, I define the defunctionalization transformation from CC_S to DCC in a similar way as that from CC to DCC – a term transformation $[\![-]\!]$ and a meta-function $[\![-]\!]_d$ for extracting definitions. Next, I show that σ and defunctionalization for 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, I show that the following diagram commutes for all CC-terms e_1 and e_2 (contexts omitted).

$$\begin{tikzcd}
e_1 \arrow[rr, "\triangleright^*"] \arrow[d, "\sigma"] \arrow[dd, bend right, "{[\![-]\!]}"'] & & e_2 \arrow[d, "\sigma"'] \arrow[dd, bend left, "{[\![-]\!]}"] \\
e_1 \arrow[r, "\triangleright^*"] \arrow[d, "{[\![-]\!]}"] & e \arrow[r, "\equiv"] \arrow[d, "{[\![-]\!]}"] & e_2 \arrow[d, "{[\![-]\!]}"'] \\
\mathsf{e}_1 \arrow[r, "\triangleright^*"] & \mathsf{e} \arrow[r, "\equiv"] & \mathsf{e}_2
\end{tikzcd} \tag{5.1}$$

CC_S is an extension of CC with new syntax, type derivation rules, reduction rules, and equivalence rules (Figure 5.1). I write CC_S expressions in a *teal, mathematical font* to avoid ambiguity. CC_S extends the CC syntax with *syntactic substitutions* of the form $e_1\{x \mapsto e_2\}$.

Type rules for variables, universes, Π-types, functions, and equivalence in CC_S are the same as the standard rules in CC, except that the type of an application $e_1\ e_2$ is $B\{x \mapsto e2\}$ with the syntactic substitution. The type of a substitution $e_1\{x \mapsto e_2\}$ is the type of e_1

with x substituted by e_2 (Subst).

CC$_S$ has five rules (those prefixed with Sub-) that reduce substitutions, which are the standard meta-theoretic substitution rules for variables, universes, Π-types, and applications being internalised into the language. Note that the meta-theoretic substitution in the CC's original beta-reduction rule $(\lambda x : A.e_1)\ e_2 \ \triangleright\ e_1\{x \mapsto e_2\}$ is also replaced by the syntactic one. CC$_S$ does not reduce substitutions into functions, but it β-reduces them when they are applied to arguments (Beta-Sub). I write $e\{x_1 \mapsto e_1, x_2 \mapsto e_2\}$ for a substitution followed by another substitution $(e\{x_1 \mapsto e_1\})\{x_2 \mapsto e_2\}$, and $e\{\bar{y} \mapsto \bar{e}\}$ for a sequence of substitutions $(((e\{y_1 \mapsto e_1\})\{x_2 \mapsto y_2\})\cdots)\{y_n \mapsto e_n\}$.

CC$_S$ has standard equivalence rules (Eq-reduce), (Eq-Eta1), and (Eq-Eta2), defined in the same way as those rules in are CC. Apart from that, it has two new symmetric rules (Eq-SubEta1) and (Eq-SubEta2) for determining when a sequence of substitutions into a function $(\lambda x : A.e)\{\bar{y} \mapsto \bar{e}\}$ is equivalent to another expression. This is essentially a variant of the η-equivalence rules that is compatible with substitutions – $(\lambda x : A.e)\{\bar{y} \mapsto \bar{e}\}$ is equivalent to e_2 if applying e_2 to x is equivalent to the function body e with $\bar{y}$ being substituted for $\bar{e}$.

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

$$\frac{\Gamma, x : A \vdash e_1 : B \rightsquigarrow \mathsf{e_1} \qquad \Gamma \vdash e_2 : A \rightsquigarrow \mathsf{e_2}}{\Gamma \vdash e_1\{x \mapsto e_2\} : B\{x \mapsto e2\} \rightsquigarrow \mathsf{e_1}[\mathsf{e_2}/\mathsf{x}]} \quad \text{(T-Subst)}$$

$$\frac{\Gamma, x : A \vdash e_1 : B \rightsquigarrow_d \mathfrak{D}_1 \qquad \Gamma \vdash e_2 : A \rightsquigarrow_d \mathfrak{D}_2}{\Gamma \vdash e_1\{x \mapsto e_2\} : B\{x \mapsto e2\} \rightsquigarrow_d \mathfrak{D}_1 \cup \mathfrak{D}_2} \quad \text{(D-Subst)}$$

The transformation turns a syntactic substitution in CC$_S$ into a meta-theoretic substitution in DCC (T-Subst); the function definitions in a substitution $e_1\{x \mapsto e_2\}$ are the union of the definitions in e_1 and e_2 (D-Subst). Since substitutions into functions do not reduce in 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.

$$\Gamma \vdash e_1 \ \triangleright^* e_2 \implies [\![e_2]\!]_d \subseteq [\![\Gamma]\!]_d \cup [\![e_1]\!]_d \tag{5.2}$$

$$[\![e_1\{x \mapsto e_2\}]\!] = [\![e_1]\!][[\![e_2]\!]/\mathsf{x}]. \tag{5.3}$$

Next, I show that the transformation preserves small step reductions in CC$_S$ – if a CC$_S$ program e_1 reduces to e_2 in one step, then the translated program $\mathsf{e_1}$ evaluates to $\mathsf{e_2}$ in a sequence.

$$\textit{Expressions} \quad ::= \quad \cdots \mid e_1\{x \mapsto e_2\}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : A.B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B\{x \mapsto e2\}} \quad \text{(Apply)}$$

$$\frac{\Gamma, x : A \vdash e_1 : B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\{x \mapsto e_2\} : B\{x \mapsto e2\}} \quad \text{(Subst)}$$

$$\begin{array}{rcll}
x\{y \mapsto e\} & \triangleright & x & \text{(Sub-Var1)} \\
x\{x \mapsto e\} & \triangleright & e & \text{(Sub-Var2)} \\
U_i\{x \mapsto e\} & \triangleright & U_i & \text{(Sub-Universe)} \\
(e_1\ e_2)\{x \mapsto e\} & \triangleright & (e_1\{x \mapsto e\})\ (e_2\{x \mapsto e\}) & \text{(Sub-Apply)} \\
(\Pi x : A.B)\{y \mapsto e\} & \triangleright & \Pi x : A\{y \mapsto e\}.B\{y \mapsto e\} & \text{(Sub-Pi)} \\
(\lambda x : A.e_1)\ e_2 & \triangleright & e_1\{x \mapsto e_2\} & \text{(Beta)} \\
((\lambda x : A.e_1)\{\bar{y} \mapsto \bar{e}\})\ e_2 & \triangleright & e_1\{\bar{y} \mapsto \bar{e}, x \mapsto e_2\} & \text{(Beta-Sub)}
\end{array}$$

$$\frac{\begin{array}{ll} e_1 \triangleright^* (\lambda x : A.e)\{\bar{y} \mapsto \bar{e}\} & e_2 \triangleright^* e_2' \\ \Gamma, x : A\{\bar{y} \mapsto \bar{e}\} \vdash e\{\bar{y} \mapsto \bar{e}\} \equiv e_2'\ x & \end{array}}{\Gamma \vdash e_1 \equiv e_2} \quad \text{(Eq-SubEta1)}$$

$$\frac{\begin{array}{ll} e_2 \triangleright^* (\lambda x : A.e)\{\bar{y} \mapsto \bar{e}\} & e_1 \triangleright^* e_1' \\ \Gamma, x : A\{\bar{y} \mapsto \bar{e}\} \vdash e_1'\ x \equiv e\{\bar{y} \mapsto \bar{e}\} & \end{array}}{\Gamma \vdash e_1 \equiv e_2} \quad \text{(Eq-SubEta2)}$$

Figure 5.1: New syntax and rules in CC$_S$

Lemma 5.1.3 (Preservation of small step reductions). If $\Gamma \vdash e_1 \triangleright e_2$, then $[\![\Gamma]\!]_d \cup [\![e_1]\!]_d ; \Gamma \vdash$ $e_1 \triangleright^* e_2$.

Proof. By induction on the reduction rules of CC_S. All cases are trivial except for the two β-reduction rules. In case (Beta), the assumption is $(\lambda^i x : A.e_1)\ e_2 \triangleright e_1\{x \mapsto e_2\}$, and the goal is to show that $\mathfrak{D} ; \Gamma \vdash \mathsf{Apply}\ \mathfrak{L}_i\{\bar{x}\}\ e_2 \triangleright^* e_1[e_2/x]$, where $\mathfrak{D} = [\![\Gamma]\!]_d \cup [\![e_1]\!]_d$ and $\bar{x}$ corresponds to the free variables in λ^i. By definition of $[\![-]\!]_d$, we have $\mathfrak{L}_i \mapsto (\{\bar{x} : \_\},\ \_,\ e_1) \in \mathfrak{D}$ (ignoring type information). So, $\mathfrak{D} ; \Gamma \vdash \mathsf{Apply}\ \mathfrak{L}_i\{\bar{x}\}\ e_2 \triangleright e_1[\bar{x}/\bar{x},\ e_2/x] = e_1[e_2/x]$. Rule (Beta-Sub) follows similarly. $\square$

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

Lemma 5.1.4 (Preservation of reduction sequences (CC_S)). If $\Gamma \vdash e_1 \triangleright^* e_2$, then $[\![\Gamma]\!]_d \cup [\![e_1]\!]_d ; \Gamma \vdash e_1 \triangleright^* e_2$.

The transformation is also coherent, i.e. it preserves the equivalence relation in CC_S.

Lemma 5.1.5 (Coherence (CC_S)). If $\Gamma \vdash e_1 \equiv e_2$, then $\mathfrak{D} ; \Gamma \vdash e_1 \equiv e_2$, where $\mathfrak{D} = [\![\Gamma]\!]_d \cup [\![e_1]\!]_d \cup [\![e_2]\!]_d$.

Proof. By induction on the equivalence rules of CC_S. Only the proof steps for the case (Eq-Eta1) are shown. Case (Eq-SubEta1) follows from similar (and more tedious) proof steps as case (Eq-Eta1), and cases (Eq-Eta2) and (Eq-SubEta2) are true by symmetry. Case (Eq-reduce) follows directly from the preservation of reduction sequences.

If $\Gamma \vdash e_1 \equiv e_2$ by (Eq-Eta1) in CC_S, then $e_1 \triangleright^* (\lambda^i x : A.e)$, $e_2 \triangleright^* e_2'$, and $\Gamma, x : A \vdash e \equiv e_2'\ x$. By Lemma 5.1.4, we have $\mathfrak{D}_1 ; \Gamma \vdash e_1 \triangleright^* \mathfrak{L}_i\{\bar{x}\}$ and $\mathfrak{D}_2 ; \Gamma \vdash e_2 \triangleright^* e_2'$, where $\mathfrak{D}_1 = [\![\Gamma]\!]_d \cup [\![e_1]\!]_d$, $\mathfrak{L}_i \mapsto (\{\bar{x} : \_\},\ \_,\ e) \in \mathfrak{D}_1$ (ignoring type information), and $\mathfrak{D}_2 = [\![\Gamma]\!]_d \cup [\![e_2]\!]_d$. By the induction hypothesis, $\mathfrak{D}_3 ; \Gamma,\ x : A \vdash e \equiv \mathsf{Apply}\ e_2'\ x$, where $\mathfrak{D}_3 = [\![\Gamma, x : A]\!]_d \cup [\![e]\!]_d \cup [\![e_2']\!]_d$. The goal in this case is to show $\mathfrak{D} ; \Gamma \vdash e_1 \equiv e_2$ by using DCC's (Eq-Eta1) rule (shown below).

$$\frac{\begin{array}{l}\mathfrak{D} ; \Gamma \vdash e_1 \triangleright^* \mathfrak{L}\{\bar{e}\} \qquad \mathfrak{D} ; \Gamma \vdash e_2 \triangleright^* e_2' \\ \mathfrak{L} \mapsto (\{\bar{x} : \bar{A}\},\ \mathsf{Pi}(x, A, B),\ e) \in \mathfrak{D} \\ \mathfrak{D} ; \Gamma,\ x : A[\bar{e}/\bar{x}] \vdash e[\bar{e}/\bar{x}] \equiv \mathsf{Apply}\ e_2'\ x\end{array}}{\mathfrak{D} ; \Gamma \vdash e_1 \equiv e_2}$$

We already have all the premises like $e_1 \triangleright^* \mathfrak{L}_i\{\bar{x}\}$, etc., but they are not judged in the desired label context $\mathfrak{D} = [\![\Gamma]\!]_d \cup [\![e_1]\!]_d \cup [\![e_2]\!]_d$. We only need to show that $\mathfrak{D}_1$, $\mathfrak{D}_2$, and $\mathfrak{D}_3$ are subsets of $\mathfrak{D}$ and to apply the weakening theorem. $\mathfrak{D}_1$ and $\mathfrak{D}_2$ are clearly subsets of

$\mathfrak{D}$, and $\mathfrak{D}_3 = [\![\Gamma]\!]_d \cup [\![A]\!]_d \cup [\![e]\!]_d \cup [\![e'_2]\!]_d$ since $[\![-]\!]_d$ acts pointwise on contexts. Also,

$$\begin{aligned} [\![A]\!]_d \cup [\![e]\!]_d &\subseteq [\![(\lambda^i x : A.e)]\!]_d \text{ by def.} \\ &\subseteq [\![\Gamma]\!]_d \cup [\![e_1]\!]_d \text{ by (5.2),} \end{aligned}$$

and $[\![e'_2]\!]_d \subseteq [\![\Gamma]\!]_d \cup [\![e_2]\!]_d$ by (5.2). Therefore, $\mathfrak{D}_3$ is also a subset of $\mathfrak{D}$. □

I use σ to denote the trivial transformation from CC to CC$_S$. This trivial transformation commutes with the two term transformations by definition.

$$[\![\sigma(e)]\!] = [\![e]\!] \tag{5.4}$$

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

$$[\![\sigma(e)]\!]_d \subseteq [\![e]\!]_d \tag{5.5}$$

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

Lemma 5.1.6 (Preservation of reduction sequences (σ))**.** If $\Gamma \vdash e_1 \ \triangleright^* e_2$, then $\Gamma \vdash e_1 \ \triangleright^* e$ where $\Gamma \vdash e \equiv e_2$.

Proof. Firstly, if $\Gamma, x : A \vdash e_1 : B$ and $\Gamma \vdash e_2 : A$, then $\Gamma \vdash \sigma(e_1[e_2/x]) \equiv e_1\{x \mapsto e_2\}$. This can be proved by induction on the type derivations of e_1, and all cases are trivial.

Therefore, we have $(\lambda x : A.e_1)\ e_2 \ \triangleright \ e_1\{x \mapsto e_2\} \equiv \sigma(e_1[e_2/x])$, so σ preserves small step reductions. Again, the preservation of reduction sequences follows immediately from this. □

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

Lemma 5.1.7 (Preservation of reduction sequences)**.** For all e_1 and e_2, $\Gamma \vdash e_1 \ \triangleright^* e_2$ implies that

$$(\mathfrak{D}_\Gamma \cup \mathfrak{D}_{e_1} \cup \mathfrak{D}_{e_2}); \Gamma \vdash e_1 \triangleright^* e, \tag{5.6}$$

$$(\mathfrak{D}_\Gamma \cup \mathfrak{D}_{e_1} \cup \mathfrak{D}_{e_2}); \Gamma \vdash e \equiv e_2 \tag{5.7}$$

for some e where $(\mathfrak{D}_\Gamma, \mathfrak{D}_{e_1}, \mathfrak{D}_{e_2}) = ([\![\Gamma]\!]_d, [\![e_1]\!]_d, [\![e_2]\!]_d)$ and $(\Gamma, e_1, e_2) = ([\![\Gamma]\!], [\![e_1]\!], [\![e_2]\!])$.

Proof. Suppose that $\Gamma \vdash e_1 \ \triangleright^* e_2$. Then, we have $\Gamma \vdash e_1 \ \triangleright^* e$ and $\Gamma \vdash e \equiv e_2$ for some e, since σ preserves reduction sequences. By Lemma 5.1.4 and Lemma 5.1.5,

$$(\llbracket\Gamma\rrbracket_d \cup \llbracket e_1\rrbracket_d)\,;\Gamma \vdash \mathsf{e}_1 \triangleright^* \mathsf{e}$$
$$(\llbracket\Gamma\rrbracket_d \cup \llbracket e_1\rrbracket_d \cup \llbracket e_2\rrbracket_d)\,;\Gamma \vdash \mathsf{e} \equiv \mathsf{e}_2.$$

Finally, since $(\llbracket\Gamma\rrbracket_d \cup \llbracket e_1\rrbracket_d)$ and $(\llbracket\Gamma\rrbracket_d \cup \llbracket e_1\rrbracket_d \cup \llbracket e_2\rrbracket_d)$ are both subsets of $(\mathfrak{D}_\Gamma \cup \mathfrak{D}_{\mathsf{e}_1} \cup \mathfrak{D}_{\mathsf{e}_2})$, we have $(\mathfrak{D}_\Gamma \cup \mathfrak{D}_{\mathsf{e}_1} \cup \mathfrak{D}_{\mathsf{e}_2})\,;\Gamma \vdash \mathsf{e}_1 \triangleright^* \mathsf{e}$ and $(\mathfrak{D}_\Gamma \cup \mathfrak{D}_{\mathsf{e}_1} \cup \mathfrak{D}_{\mathsf{e}_2})\,;\Gamma \vdash \mathsf{e} \equiv \mathsf{e}_2$ by weakening. □

Since ground types and values do not contain functions, $\llbracket v\rrbracket_d = \cdot$, and the correctness of the transformation is just a special case of Lemma 5.1.7.

Corollary 5.1.8. (Correctness) For all ground types A and values v of type A,

$$\cdot \vdash e : A \ \wedge \ \cdot \vdash e \ \triangleright^* v \implies (\mathfrak{D}_\Gamma \cup \mathfrak{D}_{\mathsf{e}})\,;\cdot \vdash \mathsf{e} \ \triangleright^* \mathsf{v}' \text{ where } \mathsf{v}' \equiv \mathsf{v}.$$

I now present the proof of type-preservation, which requires three lemmas: *substitution*, *preservation of reduction sequences*, and *coherence*. I proved that dependently-typed defunctionalization preserves reduction sequences in Lemma 5.1.7 with the help of CC$_S$, and now I prove the remaining two lemmas in a similar way. The substitution lemma states that defunctionalization is compatible with substitutions.

Lemma 5.1.9 (Substitution)**.** If $\Gamma, x:A \vdash e_1 : B$ and $\Gamma \vdash e_2 : A$, then $\mathfrak{D}\,;\Gamma \vdash \llbracket e_1[e_2/x]\rrbracket \equiv \mathsf{e}_1[\mathsf{e}_2/\mathsf{x}]$, where $\mathfrak{D} = \mathfrak{D}_\Gamma \cup \mathfrak{D}_{\mathsf{e}_1} \cup \mathfrak{D}_{\mathsf{e}_2} \cup \mathfrak{D}_{\mathsf{e}_1[\mathsf{e}_2/\mathsf{x}]}$.

Proof. From the proof of Lemma 5.1.6, we know that $\Gamma, x:A \vdash e_1 : B$ and $\Gamma \vdash e_2 : A$ implies that $\Gamma \vdash \sigma(e_1[e_2/x]) \equiv e_1\{x \mapsto e_2\}$. This further implies that $\mathfrak{D}'\,;\llbracket\Gamma\rrbracket \vdash \llbracket\sigma(e_1[e_2/x])\rrbracket \equiv \llbracket e_1\{x \mapsto e_2\}\rrbracket$, where $\mathfrak{D}' = \llbracket\Gamma\rrbracket_d \cup \llbracket e_1\rrbracket_d \cup \llbracket e_2\rrbracket_d \cup \llbracket\sigma(e_1[e_2/x])\rrbracket_d$.

By (5.5), $\mathfrak{D}' \subseteq \mathfrak{D}$. Also, $\llbracket\sigma(e_1[e_2/x])\rrbracket = \llbracket e_1[e_2/x]\rrbracket$, $\llbracket\Gamma\rrbracket = \Gamma$, and $\llbracket e_1\rrbracket[\llbracket e_2\rrbracket/\mathsf{x}] = \mathsf{e}_1[\mathsf{e}_2/\mathsf{x}]$ by (5.3) and (5.4). Therefore, we have $\mathfrak{D}\,;\Gamma \vdash \llbracket e_1[e_2/x]\rrbracket \equiv \mathsf{e}_1[\mathsf{e}_2/\mathsf{x}]$ by weakening. □

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

Lemma 5.1.10 (Coherence)**.** If $\Gamma \vdash e_1 \equiv e_2$, then $\mathfrak{D}\,;\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2$, where $\mathfrak{D} = \mathfrak{D}_\Gamma \cup \mathfrak{D}_{\mathsf{e}_1} \cup \mathfrak{D}_{\mathsf{e}_2}$.

Proof. If $\Gamma \vdash e_1 \equiv e_2$, then we have $\Gamma \vdash e_1 \equiv e_2$ by definition, since the source language never contain explicit substitutions. By Lemma 5.1.5, we have $\mathfrak{D}'\,;\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2$, where $\mathfrak{D}' = \llbracket\Gamma\rrbracket_d \cup \llbracket e_1\rrbracket_d \cup \llbracket e_2\rrbracket_d$. Since $\mathfrak{D}' \subseteq \mathfrak{D}$, we have $\mathfrak{D}\,;\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2$ by weakening. □

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

Theorem 5.1.11 (Type preservation)**.** For all well-typed programs $\Gamma \vdash e : A$,

$$\Gamma \vdash e : A \Longrightarrow (\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}}) ; \mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{A},$$

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

Proof. By proving the following two statements together with a simulteneous induction on mutually-defined judgements $\vdash \Gamma$ and $\Gamma \vdash e : A$.

1. $\vdash \Gamma \Longrightarrow \vdash \mathfrak{D}_{\mathsf{\Gamma}} ; \mathsf{\Gamma}$.

2. $\Gamma \vdash e : A \Longrightarrow (\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}}) ; \mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{A}$.

Statement 1 follows trivially from the inductive hypothesis. For statement 2, cases (Var), (Universe), and (Pi) are trivial by induction, and (Equiv) follows directly from the coherence lemma. Cases (Apply) and (Lambda) require some efforts to prove.

Case (Apply). Suppose $\Gamma \vdash e_1\ e_2 : B[e_2/x]$. The goal in this case is to show $\mathfrak{D} ; \mathsf{\Gamma} \vdash \mathsf{Apply}\ \mathsf{e}_1\ \mathsf{e}_2 : \llbracket B[e_2/x] \rrbracket$, where $\mathfrak{D} = (\mathfrak{D}_{\mathsf{\Gamma}} \cup \llbracket e_1\ e_2 \rrbracket_d)$. We have $(\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}_1}) \vdash \mathsf{e}_1 : \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})$ and $(\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}_2}) \vdash \mathsf{e}_2 : \mathsf{B}$ by the induction hypothesis. By (Apply) and weakening, $\mathfrak{D} ; \mathsf{\Gamma} \vdash \mathsf{Apply}\ \mathsf{e}_1\ \mathsf{e}_2 : \mathsf{B}[\mathsf{e}_2/\mathsf{x}]$. By the substitution lemma, $\mathfrak{D}' ; \mathsf{\Gamma} \vdash \mathsf{B}[\mathsf{e}_2/\mathsf{x}] \equiv \llbracket B[e_2/x] \rrbracket$, where $\mathfrak{D}' = (\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}_2} \cup \mathfrak{D}_{\mathsf{B}} \cup \mathfrak{D}_{\mathsf{B}[\mathsf{e}_2/\mathsf{x}]})$. We have the goal if $\mathfrak{D}' \subseteq \mathfrak{D}$. Indeed,

1. $\mathfrak{D}_{\mathsf{\Gamma}} \subseteq \mathfrak{D}$ by def.

2. $\mathfrak{D}_{\mathsf{e}_2} \subseteq \mathfrak{D}$ since $\mathfrak{D}_{\mathsf{e}_2} \subseteq \llbracket e_1\ e_2 \rrbracket_d$ by def.

3. $\mathfrak{D}_{\mathsf{B}[\mathsf{e}_2/\mathsf{x}]} \subseteq \mathfrak{D}$ since $\mathfrak{D}_{\mathsf{B}[\mathsf{e}_2/\mathsf{x}]} \subseteq \llbracket e_1\ e_2 \rrbracket_d$ by def.

4. $\mathfrak{D}_{\mathsf{B}} \subseteq \mathfrak{D}$, because $\mathfrak{D}_{\mathsf{B}} \subseteq \llbracket \Pi x : A.\, B \rrbracket$ by def., $\llbracket \Pi x : A.\, B \rrbracket \subseteq \mathfrak{D}_{\mathsf{e}_1}$ by $\Gamma \vdash e_1 : \Pi x : A.\, B$ and Lemma 4.3.5, and $\mathfrak{D}_{\mathsf{e}_1} \subseteq \mathfrak{D}$ by def.

Case (Lambda). Suppose $\Gamma \vdash \lambda^i x : A.\ e : \Pi x : A.\, B$. The goal here is to show that $\mathfrak{D} ; \mathsf{\Gamma} \vdash \mathfrak{L}_{\mathsf{i}}\{\bar{\mathsf{x}}\} : \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})$, where $\mathfrak{D} = (\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}})$, $\mathfrak{L}_{\mathsf{i}} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\},\ \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}),\ \mathsf{e})$. We have $\Gamma \vdash \bar{x} : \bar{A}$ since $\bar{x}$ are well-typed free variables, therefore, we have $\mathfrak{D} ; \mathsf{\Gamma} \vdash \bar{\mathsf{x}} : \bar{\mathsf{A}}$ by the inductive hypothesis and weakening. We also have $\mathfrak{L}_{\mathsf{i}}$ in $\mathfrak{D}$ by definition. If $\vdash \mathfrak{D} ; \mathsf{\Gamma}$ is true, the goal can be shown with DCC's type rule (Label). $\vdash (\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}}) ; \mathsf{\Gamma}$ can be obtained from the induction hypothesis and $\Gamma, x : A \vdash e : B$. Letting $\mathsf{\Gamma}_{\mathsf{fv}} = \cdot,\ \bar{\mathsf{x}} : \bar{\mathsf{A}}$, if the following statements are true, then we have $\vdash \mathfrak{D} ; \mathsf{\Gamma}$ by ($\mathsf{WFL\text{-}Label}$) and weakening.

(a) $(\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}}) ; \mathsf{\Gamma}_{\mathsf{fv}} \vdash \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}) : \mathsf{U}$.

(b) $(\mathfrak{D}_{\mathsf{\Gamma}} \cup \mathfrak{D}_{\mathsf{e}}) ; \mathsf{\Gamma}_{\mathsf{fv}},\ \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B}$.

By Lemma 4.3.2, $\mathrm{FV}(\lambda^i x : A.\ e) \vdash (\lambda^i x : A.\ e) : \Pi x : A.\, B$, so $\mathrm{FV}(\lambda^i x : A.\ e) \vdash \Pi x : A.\, B : U$ and $\mathrm{FV}(\lambda^i x : A.\ e), x : A \vdash e : B$. Therefore, conditions (a) and (b) are true by the induction hypothesis and weakening. □

$$\frac{}{\mathfrak{D}\,;\Gamma \vdash \mathsf{x} : \mathsf{A} \rightsquigarrow_b x} \qquad \text{(B-Var)}$$

$$\frac{}{\mathfrak{D}\,;\Gamma \vdash \mathsf{U_i} : \mathsf{U_{i+1}} \rightsquigarrow_b U_i} \qquad \text{(B-Universe)}$$

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

$$\frac{\mathfrak{D}\,;\Gamma \vdash \mathsf{e_1} : \mathsf{Pi(x,A,B)} \rightsquigarrow_b e_1 \qquad \Gamma \vdash \mathsf{e_2} : \mathsf{A} \rightsquigarrow_b e_2}{\mathfrak{D}\,;\Gamma \vdash \mathsf{Apply}\ \mathsf{e_1}\ \mathsf{e_2} : : \mathsf{B[e2/x]} \rightsquigarrow_b e_1\ e_2} \qquad \text{(B-Apply)}$$

$$\frac{\begin{array}{ll} \mathfrak{L} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\},\ \mathsf{Pi(x,A,B)},\ \mathsf{e}) \in \mathfrak{D} & \mathfrak{D}\,;\Gamma \vdash \bar{\mathsf{e}} : \bar{\mathsf{A}} \rightsquigarrow_b \bar{e} \\ \mathfrak{D}\,;\Gamma \vdash \mathsf{A} : \mathsf{U} \rightsquigarrow_b A & \mathfrak{D}\,;\Gamma,\ \mathsf{x} : \mathsf{A} \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow_b e \end{array}}{\mathfrak{D}\,;\Gamma \vdash \mathfrak{L}\{\bar{\mathsf{e}}\} : \mathsf{Pi(x,A[\bar{e}/\bar{x}],B[\bar{e}/\bar{x}])} \rightsquigarrow_b (\lambda x : A[\bar{e}/\bar{x}].\ e[\bar{e}/\bar{x}])} \qquad \text{(B-Lambda)}$$

$$\frac{\mathfrak{D}\,;\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow_b e}{\mathfrak{D}\,;\Gamma \vdash \mathsf{e} : \mathsf{B} \rightsquigarrow_b e} \qquad \text{(B-Equiv)}$$

Figure 5.2: Backward transformation

5.2 Type safety and consisntency

As a dependent type theory, DCC should be type-safe when it acts as a programming language and consistent when interpreted as a logic system. I prove these properties in this section by defining a *backward transformation* from DCC to CC. If the backward transformation preserves reduction sequences, then reducing a term in DCC is equivalent to reducing a term in CC. If the transformation is type-preserving and it turns the logical interpretation of *false* in DCC into that of CC, then valid proofs (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. In other words, I show that DCC can be modelled by CC in a consistent and meaning-preserving way. This is a standard technique in the literature [28, 11]. Type preservation for the backward transformation also requires the *substitution*, *preservation of reduction sequences*, and *coherence* lemmas, similar to the proof of Theorem 5.1.11. There is no particular difficulty in proving lemmas in this section.

I define the backward transformation $[\![-]\!]$ with a new judgement (Figure 5.2) of the form $\mathfrak{D}\,;\Gamma \vdash \mathsf{e} : \mathsf{A} \rightsquigarrow_b e$ and $[\![\mathsf{e}]\!] \triangleq e$. It transcribes variables, universes, Π-types, and applications back to their corresponding forms in CC. For a label term $\mathfrak{L}\{\bar{\mathsf{x}}\}$ where $\mathfrak{L} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\},\ \mathsf{Pi(x,A,B)},\ \mathsf{e})$, the transformation turns it into $(\lambda x : A[\bar{e}/\bar{x}].\ e[\bar{e}/\bar{x}])$ – a function with all of its free-variable values substituted in, where A, e, and $\bar{e}$ stand for $[\![\mathsf{A}]\!]$, $[\![\mathsf{e}]\!]$, and $[\![\bar{\mathsf{e}}]\!]$ respectively (B-Label). Intuitively, $[\![-]\!]$ decompiles a label back to the function it represents. The backward transformation also acts pointwise on DCC's type context.

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

Next, I show that the backward transformation is compatible with subsitutions. As a convention in this chapter, I use e to stand for $[\![\mathsf{e}]\!]$ when there is no ambiguity.

Lemma 5.2.1. If $\mathfrak{D} ; \Gamma,\ \mathsf{x} : \mathsf{A} \vdash \mathsf{e_1} : \mathsf{B}$ and $\mathfrak{D} ; \Gamma \vdash \mathsf{e_2} : \mathsf{A}$, then $[\![\mathsf{e_1}[\mathsf{e_2}/\mathsf{x}]]\!] = e_1[e_2/x]$.

Proof. By induction on the type derivation of $\mathsf{e_1}$. The only non-trivial case is (Label), all other cases follow directly from the induction hypothesis.

In case (Label), the assumptions are $\mathfrak{D} ; \Gamma,\ \mathsf{y} : \mathsf{C} \vdash \mathfrak{L}\{\bar{\mathsf{e}}\} : \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B})$, $\mathfrak{L} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\},\ \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}),\ \mathsf{e}) \in \mathfrak{D}$, and $\mathfrak{D} ; \Gamma \vdash \mathsf{e_2} : \mathsf{C}$. The goal is showing that $[\![(\mathfrak{L}\{\bar{\mathsf{e}}\})[\mathsf{e_2}/\mathsf{y}]]\!] = (\lambda x : A[\bar{e}/\bar{x}].\ e[\bar{e}/\bar{x}])[e_2/y]$. Indeed, we have

$$[\![(\mathfrak{L}\{\bar{\mathsf{e}}\})[\mathsf{e_2}/\mathsf{y}]]\!] = [\![\mathfrak{L}\{\bar{\mathsf{e}}[\mathsf{e_2}/\mathsf{y}]\}]\!] = \lambda x : A[(\bar{e}[e_2/y])/\bar{x}].\ e[(\bar{e}[e_2/y])/\bar{x}]$$

by the induction hypothesis, and

$$\begin{aligned}(\lambda x : A[\bar{e}/\bar{x}].\ e[\bar{e}/\bar{x}])[e_2/y] &= \lambda x : ((A[\bar{e}/\bar{x}])[e_2/y]).\ ((e[\bar{e}/\bar{x}])[e_2/y]) \\ &= \lambda x : A[(\bar{e}[e_2/y])/\bar{x}].\ e[(\bar{e}[e_2/y])/\bar{x}]\end{aligned}$$

since y is not free in A and e ($\bar{x}$ are all the free variables in them). □

Similar to proofs in Section 5.1, I show preservation of reduction sequences by showing that the transformation preserves small-step reductions.

Lemma 5.2.2. If $\mathfrak{D} ; \Gamma \vdash \mathsf{e_1} \rhd^* \mathsf{e_2}$, then $\Gamma \vdash e_1 \rhd^* e_2$.

Proof. Firstly, the backward transformation preservs small-step reductions. There is only one reduction rule (Beta) in DCC. Assume that $\mathfrak{D} ; \Gamma \vdash\ \mathsf{Apply}\ \mathfrak{L}\{\bar{\mathsf{e}}\}\ \mathsf{e_2}\ \rhd\ \mathsf{e_1}[\bar{\mathsf{e}}/\bar{\mathsf{x}},\ \mathsf{e_2}/\mathsf{x}]$ and $\mathfrak{L} \mapsto (\{\bar{\mathsf{x}} : \bar{\mathsf{A}}\},\ \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}),\ \mathsf{e_1}) \in \mathfrak{D}$. We have $[\![\mathsf{Apply}\ \mathfrak{L}\{\bar{\mathsf{e}}\}\ \mathsf{e_2}]\!] = (\lambda x : A[\bar{e}/\bar{x}].\ e_1[\bar{e}/\bar{x}])$, and

$$\begin{aligned}\Gamma \vdash (\lambda x : A[\bar{e}/\bar{x}].\ e_1[\bar{e}/\bar{x}])\ e_2\ \rhd\ &(e_1[\bar{e}/\bar{x}])[e_2/x] \\ &= e_1[\bar{e}/\bar{x}, e_2/x] \text{ since } x \text{ is not free in } \bar{e} \\ &= [\![\mathsf{e_1}[\bar{\mathsf{e}}/\bar{\mathsf{x}},\ \mathsf{e_2}/\mathsf{x}]]\!] \text{ by substitution.}\end{aligned}$$

Therefore, the backward transformation preserves reduction sequences by a trivial induction on the number of small steps in the sequence. □

I also need to show coherence for the backward transformation.

Lemma 5.2.3. If $\mathfrak{D}\,;\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2$, then $\Gamma \vdash e_1 \equiv e_2$.

Proof. By induction on DCC's equivalence rules. Only proof steps for case (Eq-Eta1) are shown. Case (Eq-Eta2) is symmetric to (Eq-Eta1), and (Eq-reduce) follows directly by preservation of reduction sequences.

In case (Eq-Eta1), the assumption is

$$\frac{\begin{array}{c}\mathfrak{D}\,;\Gamma \vdash \mathsf{e}_1 \triangleright^* \mathfrak{L}\{\bar{\mathsf{e}}\} \qquad \mathfrak{D}\,;\Gamma \vdash \mathsf{e}_2 \triangleright^* \mathsf{e}_2' \\ \mathfrak{L} \mapsto (\{\bar{\mathsf{x}}:\bar{\mathsf{A}}\},\ \mathsf{Pi}(\mathsf{x},\mathsf{A},\mathsf{B}),\ \mathsf{e}) \in \mathfrak{D} \\ \mathfrak{D}\,;\Gamma,\ \mathsf{x}:\mathsf{A}[\bar{\mathsf{e}}/\bar{\mathsf{x}}] \vdash \mathsf{e}[\bar{\mathsf{e}}/\bar{\mathsf{x}}] \equiv \mathsf{Apply}\ \mathsf{e}_2'\ \mathsf{x}\end{array}}{\mathfrak{D}\,;\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2}$$

and the goal is $\Gamma \vdash e_1 \equiv e_2$. By Lemma 5.2.2, $\Gamma \vdash e_1 \triangleright^* (\lambda x : A[\bar{e}/\bar{x}].\ e[\bar{e}/\bar{x}])$ and $\Gamma \vdash e_2 \triangleright^* e_2'$. By the induction hypothesis and the substitution lemma,
$\Gamma, x : A \vdash e[\bar{e}/x] \equiv e_2'\ x$. Therefore, $\Gamma \vdash e_1 \equiv e_2$ by CC's equivalence rule (Eq-Eta1). □

The final lemma I need is type preservation.

Lemma 5.2.4. If $\mathfrak{D}\,;\Gamma \vdash \mathsf{e} : \mathsf{A}$, then $\Gamma \vdash e : A$.

Proof. By proving the following two statements together with a simulteneous induction on mutually-defined judgements $\vdash \mathfrak{D}\,;\Gamma$ and $\mathfrak{D}\,;\Gamma \vdash \mathsf{e} : \mathsf{A}$.

1. $\vdash \mathfrak{D}\,;\Gamma \implies \vdash \Gamma$.
2. $\mathfrak{D}\,;\Gamma \vdash \mathsf{e} : \mathsf{A} \implies \Gamma \vdash e : A$.

Statement 1 follows trivially from the inductive hypothesis. For statement 2, cases (Var), (Universe), and (Pi) are trivial by induction. (Equiv) follows directly from the coherence lemma, and cases (Apply) and (Label) are proved easily with the substitution lemma. □

As a corollary of the lemmas shown in this section, DCC is type-safe and consistent since CC is.

Chapter 6

Conclusion and future work

This dissertation studied type-preserving defunctionalization for a variant of the Calculus of Constructions. I illustrated that Pottier and Gauthier's type-preserving polymorphic defunctionalization does not extend to dependently-typed languages. Then, I presented abstract defunctionalization as an alternative method. Abstract defunctionalization consists of a target language (the Defunctionalized Calculus of Constructions) and a transformation from the source language to the target language. The target language has a primitive notion of function labels that fits the abstract description of defunctionalization. I proved the transformation type-preserving and correct, and I showed that the target language is type-safe as a programming language and consistent as a logic.

In this chapter, I first cover serval topics and issues in implementing the target language, then I end with a discussion on future work.

6.1 Implementation of abstract defunctionalization

This section briefly introduces my implementation of the Defunctionalized Calculus of Constructions (DCC) and the transformation from CC to DCC in OCaml. The implementation allowed me to check mechanically for complicated and counter-intuitive examples of dependently-typed defunctionalization, which are difficult and tedious to check by hand.

Syntax The code structure is based on Bauer's tutorial of implementing dependent type theory in OCaml [29]. The abstract syntax of DCC is defined with the following inductive type `expr`.

```
type expr =
    | Var of variable | Universe of int
    | Label of label * expr list
    | Apply of expr * expr
```

```
        | Pi of variable * expr * expr
        | Unit | UnitType
    type context = {
        def : (label * defItem) list; (* The label context *)
        con : (variable * expr) list; (* The type context  *)
    }
```

The data types above are the exact resemblance to DCC's syntax in Figure 4.1, and it includes the unit type as a base type. Types `variable` and `label` are string-integer pairs – the string is the name of the variable or label, and the integer is used for distinguishing variables or labels named after the same string. Data type `defItem` is a record for keeping the data associated with a label – the list of free variables, etc. (definition omitted for simplicity). I implemented utility functions `subst`, `normalize`, and `equal` to perform substitutions, normalize terms, and determine whether two terms are equivalent.

```
    subst : (variable * expr) list -> expr -> expr
    normalize : context -> expr -> expr
    equal : context -> expr -> expr -> bool
```

Type-checking and transformation In DCC, an expression has at most one type (up to equivalence), meaning that given a context and a well-typed expression, the type of the expression can be worked out algorithmically according to the type rules. I implemented function `infer_type` for inferring the type of a well-typed term and function `type_check` for checking whether a given type judgement is derivable.

```
    infer_type : context -> expr -> expr
    type_check : context -> expr -> expr -> bool
```

Given inputs `(con, e, t)`, `type_check` firstly checks if the context `con` is well-formed. Then, it computes `t' = (infer_type con e)` and checks if `t'` is equivalent to `t`.

The defunctionalization transformation is implemented with function `transform_full`. Given a type judgement `(con, e, t)` in CC (it corresponds to $\Gamma \vdash e : A$), `transform_full` firstly assigns a unique integer-valued tag to each lambda abstraction in `(con, e, t)`. Then, it computes the term transformation and extracts function definitions for the labelled CC judgement according to the rules defined in Figure 4.5 and Figure 4.6. It returns a triple `(con', e', t')`, which are the transformed context, term, and type. I can use `(type_check con' e' t')` to see if the transformation preserves types for a particular input.

Checking well-formedness DCC's type rules presented in Figure 4.2 are not suitable for practical implementation, since it checks the well-formedness of the context in every sub-derivation tree of variables, universes, and labels. This is very inefficient. Since there is no way to extend the label context in a DCC program, `type_check` only needs to check its well-formedness once at the beginning of the type-checking process. Similarly, it only checks the well-formedness of the type context once, and makes sure that whenever it extends the type context with a variable-expression pair, that expression is a valid type in the old context.

6.2 Future work

Recursive functions I plan to adapt abstract defunctionalization to the Calculus of Inductive Constructions (CIC), the extension of CC with inductive families and recursive functions. This involves defining a syntax for inductive families, pattern matching, and recursions in the target language DCC. While recursive functions do not add extra difficulty to the defunctionalization transformation [3], adding recursions to DCC is a challenge. Dependently-typed languages only accept terminating functions, since non-terminating functions make the type system inconsistent. Languages like Agda and Coq have *syntactic conditions* to guarantee termination of recursive functions. DCC should be extended with syntactic termination conditions for recursive functions, and defunctionalization should transform the termination conditions from the source language to those in the target language.

Embedding DCC into Agda It is possible to formalize DCC as an embedding in a proof assistant like Agda. The benefit of doing so is having an *intrinsic* abstract syntax that only contains well-typed terms, and the meta-theoretic properties of DCC can be formally studied in a machine-checked setting. However, dealing with variable binding and substitutions in the meta-theory is challenging. A promising tool for formalizing variable binding is the second-order abstract syntax [30], but it only applies to simply-typed systems at the moment. Other examples of embedding a dependent theory into another used quotient-inductive types [31] or shallow embeddings [32], but it is not clear how to adapt these methods to model DCC's label context.

Type-preserving compilers for dependent types Ager et al. showed that closure conversion, continuation-passing style transformation (CPS), and defunctionalization could be combined to derive compilers and virtual machines of the untyped lambda calculus [33]. Dependently-typed closure conversion and the CPS transformation are available in the literature [10, 11]. With abstract defunctionalization, the next step is to investigate whether the derivation method by Ager et al. leads to a type-preserving compiler for dependently-typed languages, which makes checking the correctness of separately-compiled and linked programs possible.

Bibliography

[1] William J Bowman. *Compiling with Dependent Types.* PhD thesis, Northeastern University, 2019.

[2] 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.

[3] 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.

[4] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 42–54. ACM, 2006.

[5] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 595–608. ACM, 2015.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM, 1972.

[13] Wei-Ngan Chin and John Darlington. A higher-order removal method. *LISP Symb. Comput.*, 9(4):287–322, 1996.

[14] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *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*, volume 1782 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2000.

[15] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 166–178. ACM, 2001.

[16] Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2014.

[17] Andrew P. Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *J. Funct. Program.*, 8(4):367–412, 1998.

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

[19] Olivier Danvy and Lasse R. Nielsen. 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*, pages 162–174. ACM, 2001.

[20] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 420–447. Springer, 2001.

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

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

[23] The Agda team. The Agda user manual. `https://agda.readthedocs.io/en/v2.6.2.1/index.html`, 2021.

[24] The Coq development team. The Coq reference manual. `https://coq.inria.fr/distrib/current/refman/`, 2021.

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

[26] 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.

[27] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.

[28] 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.

[29] Andrej Bauer. How to implement dependent type theory. `http://math.andrej.com/2012/11/08/how-to-implement-dependent-type-theory-i/`, 2012.

[30] Marcelo Fiore and Dmitrij Szamozvancev. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.*, 6(POPL):1–29, 2022.

[31] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016.

[32] Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow embedding of type theory is morally correct. *CoRR*, abs/1907.07562, 2019.

[33] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. *BRICS Report Series*, 10(14), 2003.