

Type-Level Computation One Step at a Time

Foo Bar Baz

The University of Foo
{foo,bar,baz}@foo.edu

Abstract

Many type systems support a conversion rule that allows type-level computation. In such type systems ensuring the *decidability* of type checking requires type-level computation to terminate. For calculi where the syntax of types and terms is the same, the decidability of type-checking is usually dependent on the strong normalization of the calculus, which ensures termination. An unfortunate consequence of this coupling between decidability and strong normalization is that adding (unrestricted) general recursion to such calculi is not possible.

This paper proposes an alternative to the conversion rule that allows the same syntax for types and terms, type-level computation, and preserves decidability of type-checking under the presence of general recursion. The key idea, which is inspired by the traditional treatment of *iso-recursive types*, is to make each type-level computation step explicit. Each beta reduction or expansion at the type-level is introduced by a language construct. This allows control over the type-level computation and ensures decidability of type-checking even in the presence of non-terminating programs at the type-level. We realize this idea by presenting a variant of the calculus of constructions with general recursion and recursive types. Furthermore we show how many advanced programming language features of state-of-the-art functional languages (such as Haskell) can be encoded in our minimalistic core calculus.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms Languages, Design

Keywords Dependent types, Intermediate language

1. Introduction

Modern statically typed functional languages (such as ML, Haskell, Scala or OCaml) have increasingly expressive type systems. Often these large source languages are translated into a much smaller typed core language. The choice of the core language is essential to ensure that all the features of the source language can be encoded. For a simple polymorphic functional language it is possible, for example, to pick a variant of System F as a core language. However, the desire for more expressive type system features puts

pressure on the core languages, often requiring them to be extended to support new features. For example, if the source language supports *higher-kinded types* or *type-level functions* then System F is not expressive enough and can no longer be used as the core language. Instead another core language that does provide support for higher-kinded types, such as System F_ω , needs to be used. However System F_ω is significantly more complex than System F and thus harder to maintain. If later a new feature, such as *kind polymorphism*, is desired the core language may need to be changed again to account for the new feature, introducing at the same time new sources of complexity.

The more expressive type systems become, the more types become similar to the terms. Therefore a natural idea is to unify terms and types. There are obvious benefits in this approach: only one syntactic level (terms) is needed; and there are much less language constructs, making the core language easier to implement and maintain. At the same time the core language becomes more expressive, giving us for free many useful language features. *Pure type systems* [] build on this observation and they show how a whole family of type systems (including System F and System F_ω) can be implemented using just a single syntactic form. With the added expressiveness it is even possible to have type-level programs expressed using the same syntax as terms as well as dependently typed programs [].

However having the same syntax for types and terms can also be problematic. If arbitrary type-level computation is allowed then type-level programs can use the same language constructs as terms. Usually type systems have a conversion rule to support type-level computation. In such type systems ensuring the *decidability* of type checking requires type-level computation to terminate. For calculi where the syntax of types and terms is the same, the decidability of type-checking is usually dependent on the strong normalization of the calculus, which ensures termination. An unfortunate consequence of this coupling between decidability and strong normalization is that adding (unrestricted) general recursion to such calculi is not possible.

Tension between decidability of type-checking, logical consistency and general recursion.

Brief summary of related work

This paper proposes an alternative to the conversion rule that allows the same syntax for types and terms, type-level computation, and preserves decidability of type-checking under the presence of general recursion. The key idea, which is inspired by the traditional treatment of *iso-recursive types*, is to make each type-level computation step explicit.

Point about the treatment of type-level computation in Haskell. Haskell's core language has type applications, but no type-level lambda. Equality is syntactic modulo alpha-conversion. This design choice was rooted in the desire to support Hindley-Milner type-inference...

Each beta reduction or expansion at the type-level is introduced by a language construct. This allows control over the type-level

computation and ensures decidability of type-checking even in the presence of non-terminating programs at the type-level. We realize this idea by presenting a variant of the calculus of constructions with general recursion and recursive types. Furthermore we show how many programming language features of state-of-the-art functional languages (such as Haskell) can be encoded in our minimalistic core calculus.

a) Motivations:

- Because of the reluctance to introduce dependent types¹, the current intermediate language of Haskell, namely System F_C [12], separates expressions as terms, types and kinds, which brings complexity to the implementation as well as further extensions [14, 15].
- Popular full-spectrum dependently typed languages, like Agda, Coq, Idris, have to ensure the termination of functions for the decidability of proofs. No general recursion and the limitation of enforcing termination checking make such languages impractical for general-purpose programming.
- We would like to introduce a simple and compiler-friendly dependently typed core language with only one hierarchy, which supports general recursion at the same time.

b) Contribution:

- A core language based on Calculus of Constructions (CoC) that collapses terms, types and kinds into the same hierarchy.
- General recursion by introducing recursive types for both terms and types by the same μ primitive.
- Decidable type checking and managed type-level computation by replacing implicit conversion rule of CoC with generalized fold/unfold semantics.
- First-class equality by coercion, which is used for encoding GADTs or newtypes without runtime overhead.
- Surface language that supports datatypes, pattern matching and other language extensions for Haskell, and can be encoded into the core language.

c) Related work:

- Henk [6] and one of its implementation [8] show the simplicity of the Pure Type System (PTS). [9] also tries to combine recursion with PTS.
- Zombie [3, 10] is a language with two fragments supporting logics with non-termination. It limits the β -reduction for congruence closure [11].
- $\Pi\Sigma$ [1] is a simple, dependently-typed core language for expressing high-level constructions². UHC compiler [7] tries to use a simplified core language with coercion to encode GADTs.
- System F_C [12] has been extended with type promotion [15] and kind equality [14]. The latter one introduces a limited form of dependent types into the system³, which mixes up types and kinds.

¹ This might be changed in the near future. See <https://ghc.haskell.org/trac/ghc/wiki/DependentHaskell/Phase1>.

² But the paper didn't give any meta-theories about the language.

³ Richard A. Eisenberg is going to implement kind equality [14] into GHC. The implementation is proposed at <https://phabricator.haskell.org/D808> and related paper is at <http://www.cis.upenn.edu/~eir/papers/2015/equalities/equalities-extended.pdf>.

2. Overview

BRUNO: Jeremy: can you give this section a go and start writing it up? I think this section should be your priority for now.

We begin this section with an informal introduction to the main features of λC_β . We show how it can serve as a simple and compiler-friendly core language with general recursion and decidable type system. The formal details are presented in §??.

2.1 Calculus of Constructions

λC_β is based on the *Calculus of Constructions* (λC) [5], which is a higher-order typed lambda calculus. One “unconventional” feature of λC is the so-called *conversion* rule as shown below:

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_2 : s \quad \tau_1 =_\beta \tau_2}{\Gamma \vdash e : \tau_2} \text{ TCC.CONV}$$

The conversion rule allows one to derive $e : \tau_2$ from the derivation of $e : \tau_1$ and the β -equality of τ_1 and τ_2 . Note that in λC , the use of this rule is implicit in that it is automatically applied during type checking to all non-normal form terms. To illustrate, let us consider a simple example. Suppose we have a built-in base type Int and

$$f \equiv \lambda x : (\lambda y : \star.y) \text{Int}.x$$

Without the conversion rule, f cannot be applied to, say 3 in λC . Given that f is actually β -convertible to $\lambda x : \text{Int}.x$, the conversion rule would allow the application of f to 3 by implicitly converting $\lambda x : (\lambda y : \star.y) \text{Int}.x$ to $\lambda x : \text{Int}.x$.

2.2 Explicit Type Conversion Rules

BRUNO: Contrast our calculus with the calculus of constructions. Explain fold/unfold.

In contrast to the implicit reduction rules of λC , λC_β makes it explicit as to when and where to convert one type to another. To achieve that, it makes type conversion explicit by introducing two operations: cast^\uparrow and cast_\downarrow .

In order to have a better intuition, let us consider the same example from §2.1. In λC_β , f 3 is intended as an ill-typed application. Instead one would like to write the application as

$$f (\text{cast}^\uparrow [(\lambda y : \star.y) \text{Int}] 3)$$

The intuition is that, cast^\uparrow is actually doing type conversion since the type of 3 is Int and $(\lambda y : \star.y) \text{Int}$ can be reduced to Int .

The dual operation of cast^\uparrow is cast_\downarrow . The use of cast_\downarrow is better explained by another similar example. Suppose that

$$g \equiv \lambda x : \text{Int}.x$$

and term z has type

$$(\lambda y : \star.y) \text{Int}$$

$g z$ is again an ill-typed application, while $g (\text{cast}_\downarrow z)$ is type correct because cast_\downarrow reduces the type of z to Int .

2.3 Decidability and Strong Normalization

BRUNO: Informally explain that with explicit fold/unfold rules the decidability of the type system does not depend on strong normalization.

The decidability of the type system of λC depends on the normalization property for all constructed terms [4]. However strong normalization does not hold with general recursion. This is simply because due to the conversion rule, any non-terminating term would force the type checker to go into an infinitely loop (by constantly applying the conversion rule without termination), thus rendering the type system undecidable.

With explicit type conversion rules, however, the decidability of the type system no longer depends on the normalization property.

In fact λC_β is not strong normalizing, as we will see in later sections. The ability to write non-terminating terms motivates us to have more control over type-level computation. To illustrate, let us consider a contrived example. Suppose that d is a “dependent type” where

$$d : \text{Int} \rightarrow \star$$

so that $d\ 3$ or $d\ 100$ all yield the same type. With general recursion at hand, we can image a term z that has type

$$d\ \text{loop}$$

where loop stands for any diverging computation and of type Int . What would happen if we try to type check the following application:

$$(\lambda x : d\ 3.x)\ z$$

Under the normal typing rules of λC , the type checker would get stuck as it tries to do β -equality on two terms: $d\ 3$ and $d\ \text{loop}$, where the latter is non-terminating.

This is not the case for λC_β : (i) it has no such conversion rule, therefore the type checker would do syntactic comparison between the two terms instead of β -equality in the above example; and (ii) one would need to write infinite number of cast_\downarrow ’s to make the type checker loop forever (e.g., $(\lambda x : d\ 3.x)(\text{cast}_\downarrow(\text{cast}_\downarrow \dots z))$, which is impossible in reality.

In summary, λC_β achieves the decidability of type checking by explicitly controlling type-level computation, which is independent of the normalization property, while supporting general recursion at the same time.

2.4 Unifying Recursive Types and Recursion

BRUNO: Show how in λC_β recursion and recursive types are unified. Discuss that due to this unification the sensible choice for the evaluation strategy is call-by-name.

Recursive types arise naturally if we want to do general recursion. λC_β differs from other programming languages in that it unifies both recursion and recursive types by the same μ primitive.

Recursive types. In the literature on type systems, there are two approaches to recursive types. One is called *equi-recursive*, the other *iso-recursive*. λC_β takes the latter approach since it is more intuitive to us with regard to recursion. The *iso-recursive* approach treats a recursive type and its unfolding as different, but isomorphic. In λC_β , this is witnessed by first cast^\uparrow , then cast_\downarrow . A classic example of recursive types is the so-called “hungry” type: $H = \mu\sigma : \star. \text{Int} \rightarrow \sigma$. A term z of type H can accept any number of numeric arguments and return a new function that is hungry for more, as illustrated below:

$$\begin{aligned} \text{cast}_\downarrow z : \text{Int} &\rightarrow H \\ \text{cast}_\downarrow(\text{cast}_\downarrow z) : \text{Int} &\rightarrow \text{Int} \rightarrow H \\ \text{cast}_\downarrow(\text{cast}_\downarrow \dots z) : \text{Int} &\rightarrow \text{Int} \rightarrow \dots \rightarrow H \end{aligned}$$

Recursion. The same μ primitive can also be used to define recursive functions, e.g., the factorial function:

$$\mu f : \text{Int} \rightarrow \text{Int}. \lambda x : \text{Int}. \text{if } (x == 0) \text{ then } 1 \text{ else } x * f\ (x - 1)$$

This is reflected by the dynamic semantics of the μ primitive:

$$\mu x : T. E \longrightarrow E[x := \mu x : T. E]$$

which is exactly doing recursive unfolding of the same term.

Due to the unification, the *call-by-value* evaluation strategy does not fit in our setting. In call-by-value evaluation, recursion can be expressed by the recursive binder μ as $\mu f : T \rightarrow T. E$ (note that the type of f is restricted to function types). Since we don’t want to pose restrictions on the types, the *call-by-name* evaluation is a sensible choice.

2.5 Encoding Datatypes

BRUNO: Informally explain how to encode recursive datatypes and recursive functions using datatypes.

With the explicit type conversion rules and the μ primitive, it is straightforward to encode recursive datatypes and recursive functions using datatypes. While inductive datatypes can be encoded using either the Church or the Scott encoding, we adopt the Scott encoding as it bears some resemblance to case analysis, making it more convenient to encode pattern matching. We demonstrate the encoding method using a simple datatype as a running example: the natural numbers.

The datatype declaration for natural numbers is:

$$\text{data } \text{Nat} = Z \mid S\ \text{Nat};$$

In the Scott encoding, the encoding of the Nat type reflects how its two constructors are going to be used. Since Nat is a recursive datatype, we have to use recursive types at some point to reflect its recursive nature. As it turns out, the Nat type can be simply represented as $\mu X : \star. \Pi B : \star. B \rightarrow (X \rightarrow B) \rightarrow B$.

As can be seen, in the function type $B \rightarrow (X \rightarrow B) \rightarrow B$, B corresponds to the type of the Z constructor, and $X \rightarrow B$ corresponds to the type of the S constructor. The intuition is that any use of the datatype being defined in the constructors is replaced with the recursive type, except for the return type, which is a type variable for use in the recursive functions.

Now its two constructors can be encoded correspondingly as below:

$$\begin{aligned} \text{let } Z : \text{Nat} &= \text{cast}^\uparrow [\text{Nat}] (\lambda B : \star. \lambda z : B. \lambda f : \text{Nat} \rightarrow B. z) \\ \text{in} \\ \text{let } S : \text{Nat} &\rightarrow \text{Nat} = \lambda n : \text{Nat}. \\ &\text{cast}^\uparrow [\text{Nat}] (\lambda B : \star. \lambda z : B. \lambda f : \text{Nat} \rightarrow B. f\ n) \\ \text{in} \end{aligned}$$

Thanks to the explicit type conversion rules, we can make use of the cast^\uparrow operation to do type conversion between the recursive type and its unfolding.

As the last example, let us see how we can define recursive functions using the Nat datatype. A simple example would be recursively adding two natural numbers, which can be defined as below:

$$\begin{aligned} \text{let } \text{add} : \text{Nat} &\rightarrow \text{Nat} \rightarrow \text{Nat} = \mu f : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}. \\ &\lambda n : \text{Nat}. \lambda m : \text{Nat}. \\ &(\text{cast}_\downarrow n)\ \text{Nat}\ m\ (\lambda n' : \text{Nat}. S\ (f\ n'\ m)) \end{aligned}$$

As we can see, the above definition quite resembles case analysis common in modern functional programming languages. (Actually we formalize the encoding of case analysis in §6.)

Due to the unification of recursive types and recursion, we can use the same μ primitive to write both recursive types and recursion with ease.

3. Applications

In this section, we show some large examples using λC_β .

3.1 Parametric HOAS

Parametric Higher Order Abstract Syntax (PHOAS) is a higher order approach to represent binders, in which the function space of the meta-language is used to encode the binders of the object language. We show that λC_β can handle PHOAS by encoding lambda calculus as below:

```

data PLambda (a :  $\star$ ) = Var a
  | Num nat
  | Lam (a  $\rightarrow$  PLambda a)
  | App (PLambda a) (PLambda a);

```

Next we define the evaluator for our lambda calculus. One advantage of PHOAS is that, environments are implicitly handled by the meta-language, thus the type of the evaluator is simply $plambda\ value \rightarrow value$. The code is presented in Figure 1.

```

data Value = VI nat
  | VF (Value  $\rightarrow$  Value);
let eval : PLambda Value  $\rightarrow$  Value =
   $\mu$  ev : PLambda Value  $\rightarrow$  Value.
   $\lambda e$  : PLambda Value. case e of
    Var (v : Value)  $\Rightarrow$  v
  | Num (n : nat)  $\Rightarrow$  VI n
  | Lam (f : Value  $\rightarrow$  PLambda Value)  $\Rightarrow$ 
    VF ( $\lambda x$  : Value. ev (f x))
  | App (a : PLambda Value) (b : PLambda Value)  $\Rightarrow$ 
    case (ev a) of
      VI (n : nat)  $\Rightarrow$  VI n -- impossible to reach
  | VF (f : Value  $\rightarrow$  Value)  $\Rightarrow$  f (ev b)
in

```

Figure 1. Lambda Calculus in PHOAS

Now we can evaluate some lambda expression and get the result back as in Figure 2

```

let show : Value  $\rightarrow$  nat =
   $\lambda e$  : Value. case e of
    VI (n : nat)  $\Rightarrow$  n
  | VF (f : Value  $\rightarrow$  Value)  $\Rightarrow$  10000 -- impossible to reach
in
let example : PLambda Value =
  App Value
    (Lam Value ( $\lambda x$  : Value. Var Value X))
    (Num Value 42)
in show (eval example) -- return 42

```

Figure 2. Example of using PHOAS

3.2 Perfect binary trees

A perfect binary tree is a binary tree whose size is exactly a power of two. In Haskell, perfect binary trees are usually represented using nested datatypes. We show that λC_β is able to encode nested datatypes.

First we define a pair datatype as follows:

```

data PairT (a :  $\star$ ) (b :  $\star$ ) = P a b;

```

Using pairs, perfect binary trees are easily defined as below:

```

data B (a :  $\star$ ) = One a | Two (B (PairT a a));

```

Notice that the recursive use of B does not hold a , but $PairT\ a\ a$. This means every time we use a Two constructor, the size of the pairs doubles. In case you are curious about the encoding of B , here is the one:

```

let B :  $\star \rightarrow \star = \mu X : \star \rightarrow \star.$ 
   $\lambda a : \star. (B : \star) \rightarrow (a \rightarrow B) \rightarrow (X (PairT\ a\ a) \rightarrow B) \rightarrow B$ 
in

```

Because of the polymorphic recursive type $(\mu X : \star \rightarrow \star)$ being used, it is fairly straightforward to encode nested datatypes.

To easily construct a perfect binary tree from a list, we define a help function that transform a list to a perfect binary tree as shown in Figure 3.

```

let pairs : (a :  $\star$ )  $\rightarrow$  List a  $\rightarrow$  List (PairT a a) =
   $\mu$  pairs' : (a :  $\star$ )  $\rightarrow$  List a  $\rightarrow$  List (PairT a a).
   $\lambda a : \star. \lambda xs : List\ a.$ 
    case xs of
      Nil  $\Rightarrow$  Nil (PairT a a)
  | Cons (y : a) (ys : List a)  $\Rightarrow$ 
    case ys of Nil  $\Rightarrow$ 
      Nil (PairT a a)
  | Cons (y' : a) (ys' : List a)  $\Rightarrow$ 
    Cons (PairT a a) (P a a y y') (pairs' a ys')
in
let fromList : (a :  $\star$ )  $\rightarrow$  List a  $\rightarrow$  B a =
   $\mu$  from' : (a :  $\star$ )  $\rightarrow$  List a  $\rightarrow$  B a.
   $\lambda a : \star. \lambda xs : List\ a.$ 
    case xs of
      Nil  $\Rightarrow$  Two a (from' (PairT a a) (pairs a (Nil a)))
  | Cons (x : a) (xs' : List a)  $\Rightarrow$ 
    case xs' of
      Nil  $\Rightarrow$  One a x
  | Cons (y : a) (zs : List a)  $\Rightarrow$ 
    Two a (from' (PairT a a) (pairs a x xs))
in

```

Figure 3. Construct a perfect binary tree from a list

Now we can define an interesting function $powerTwo$. Given a natural number n , it computes the largest natural number m , such that $2^m < n$:

```

let twos : (a :  $\star$ )  $\rightarrow$  B a  $\rightarrow$  nat =
   $\mu$  twos' : (a :  $\star$ )  $\rightarrow$  B a  $\rightarrow$  nat.
   $\lambda a : \star. \lambda x : B\ a.$ 
    case x of
      One (y : a)  $\Rightarrow$  0
  | Two (c : B (PairT a a))  $\Rightarrow$ 
    1 + twos' (PairT a a) c
in
let powerTwo : Nat  $\rightarrow$  nat =
   $\lambda n : Nat. twos\ nat\ (fromList\ nat\ (take\ n\ (repeat\ 1)))$ 
in powerTwo (S (S (S (S Z)))) -- return 2

```

4. The core language

In this section, we present λC_β calculus, the core language with explicit type-level computation. λC_β is carefully designed to be minimal enough for simplifying type checking and meta-theoretic studying, while still keeps the expressiveness to represent rich high-level constructions (§??). By explicitly controlling the type-level computation with cast operators, λC_β can safely allow non-termination without breaking the decidability of type checking. In rest of this section, we demonstrate the syntax (§4.1), type system (§4.2) and meta-theories (§4.3) of λC_β .

4.1 Syntax

Figure 4 shows the syntax of λC_β , including expressions, contexts and values. The syntax follows λC and straightforward to understand, while there are still some differences that embody the simplicity and expressiveness of λC_β .

Unified syntactic levels λC_β uses a unified syntactic representation for different levels of expressions by following the *pure type system* (PTS) representation of λC . Traditionally in λC , there are two distinct sorts \star and \square representing the type of *types* and *sorts* respectively, and an axiom $\star : \square$ specifying the relation. In λC_β , we further merge types and kinds together by including only a single sort \star and an impredicative axiom $\star : \star$.

Therefore, there is no syntactic distinction between terms, types or kinds. This design brings the economy for type checking, since one set of rules can cover all syntactic levels. By convention, we use metavariables τ and σ for an expression on the type-level position and e for one on the term level.

Dependent function types In the context of λC , if a term x has the type τ_1 , and τ_2 is a type, i.e. $x : \tau_1 : \star$ and $\tau_2 : \square$, we call the type $\Pi x : \tau_1. \tau_2$ a *dependent product*. λC_β follows λC to use the same Π -notation to represent dependent function types.

However, a higher-kind polymorphic function type such as $\Pi x : \square. x \rightarrow x$ is not allowed in λC , because \square is the highest sort that can not be typed. While Π -notation in λC_β is more expressive and does not have such limitation because of the axiom $\star : \star$. In the source language, we interchangeably use the arrow form $(x : \tau_1) \rightarrow \tau_2$ of the product for clarity. By convention, we also use the syntactic sugar $\tau_1 \rightarrow \tau_2$ to represent the product if x does not occur free in τ_2 .

General recursion We use the polymorphic recursion operator μ to represent general recursion on both term and type level in the same form $\mu x : \tau. e$. On the term level, a μ -term has the similar functionality as a fixpoint, that its unfolding $e[x \mapsto \mu x : \tau. e]$ can be achieved by one-step reduction (See examples in §??). On the type level, $\mu x : \tau. e$ represents recursive types and uses the *iso-recursive* approach, that the recursive type is not equal but only isomorphic to its unrolling.

Explicit type conversion We introduce two new primitives cast^\uparrow and cast_\downarrow (pronounced as ‘cast up’ and ‘cast down’) to replace implicit conversion rule of λC with one-step explicit type conversion. They represent two directions of type conversion: cast_\downarrow stands for the β -reduction of types, while cast^\uparrow is the inverse (see examples in §??). cast^\uparrow and cast_\downarrow also serve as the fold and unfold for iso-recursive types to map back and forth between the original and unrolled form (§4.2).

Though cast primitives make the syntax verbose when type conversion is heavily used, the implementation of type checking is simplified because typing rules of λC_β become type-directed without λC ’s implicit conversion rule. Considering the core language is compiler-oriented and **source language does not include cast primitives**, end-users will not directly use them. Some type conversions can be generated through the translation of the source language (§??).

4.2 Type system

The type system for the core language includes operational semantics and typing judgements. Typing judgements include rules of context well-formedness $\vdash \Gamma$ and expression typing $\Gamma \vdash e : \tau$. Note that there is only a single set of rules for expression typing, because of no distinction of different syntactic levels.

Operational semantics Figure 5 shows the *call-by-name* operational semantics, defined by one-step reduction relations, namely

e, τ, σ	$::=$	Expressions
	x	Variable
	\star	Type of types
	$e_1 e_2$	Application
	$\lambda x : \tau. e$	Abstraction
	$\Pi x : \tau_1. \tau_2$	Product
	$\text{cast}^\uparrow [\tau] e$	Cast up by expansion
	$\text{cast}_\downarrow e$	Cast down by reduction
	$\mu x : \tau. e$	General recursion
	$\text{cast}_\downarrow^n [\tau] e$	
	$\text{cast}_\downarrow^n e$	
Γ	$::=$	Contexts
	\emptyset	Empty
	$\Gamma, x : \tau$	Variable binding
v	$::=$	Values
	x	Variable
	\star	Type of types
	$\lambda x : \tau. e$	Abstraction
	$\Pi x : \tau_1. \tau_2$	Product
	$\text{cast}^\uparrow [\tau] e$	Cast up by expansion

Figure 4. Syntax of λC_β

$e \longrightarrow e'$	One-step reduction
$\frac{}{(\lambda x : \tau. e_1) e_2 \longrightarrow e_1[x \mapsto e_2]}$	S_BETA
$\frac{}{\mu x : \tau. e \longrightarrow e[x \mapsto \mu x : \tau. e]}$	S_MU
$\frac{}{\text{cast}_\downarrow (\text{cast}^\uparrow [\tau] e) \longrightarrow e}$	S_CASTDOWNUP
$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$	S_APP
$\frac{e \longrightarrow e'}{\text{cast}_\downarrow e \longrightarrow \text{cast}_\downarrow e'}$	S_CASTDOWN

Figure 5. Operational semantics of λC_β

\longrightarrow . Three base cases include S_BETA for β -reduction, S_MU for recursion unrolling and S_CASTDOWNUP for cast canceling. Two inductive cases, S_APP and S_CASTDOWN, define reduction in the head position of an application, and in the cast_\downarrow inner expression respectively.

The reduction rules are *weak* in the sense that it is not allowed to reduce inside a λ -term or cast^\uparrow -term which are viewed as a value (see Figure 4) that ends the reduction. Note that we do not treat the μ -term as a value, which is different from conventional iso-recursive types. Because on the term level, the μ operator is treated as a fixpoint. The μ -term is equivalent to a recursive function that should be allowed to unroll without restriction.

Expression typing Figure 6 gives the *syntax-directed* typing rules of valid context and expression. Most typing rules are quite standard. We write $\vdash \Gamma$ if a context Γ is well-formed. Since there is only a single sort \star , $\Gamma \vdash \tau : \star$ is used in rules to check if τ is a well-formed type. Rule T_AX is the ‘type-in-type’ axiom. Rule T_VAR checks the type of variable x from the valid context. Rules T_APP and T_LAM check the validity of application and abstraction. Rules

$$\overline{\mu x : \tau. e \longrightarrow e[x \mapsto \mu x : \tau. e]} \quad \text{S_MU}$$

$\mu x : \tau. e$ is the substitution of x in e with itself, i.e. $e[x \mapsto \mu x : \tau. e]$. Such behavior is just the same as the definition of a fixed point.

If $\mu x : \tau. e$ is a type, assume there exist $e_1 : \mu x : \tau. e$ and $e_2 : e[x \mapsto \mu x : \tau. e]$. Notice that the types of e_1 and e_2 are equivalent by β -equivalence. But such result cannot be directly obtained because of the removal of implicit conversion rule. Instead, by using explicit cast operations of λC_{exp} , we can obtain the following transformation between e and e' :

$$\begin{array}{ll} \text{cast}^\uparrow [\mu x : \tau. e] e_2 & : \mu x : \tau. e \\ \text{cast}_\downarrow e_1 & : (\mu x : \tau. e[x \mapsto \mu x : \tau. e]) \end{array}$$

For type-level μ -notation, cast^\uparrow and cast_\downarrow work in the same way as fold and unfold operations in iso-recursive types to control recursion explicitly.

5.2 Decidability and soundness

LINUS: Not finished. Needs thorough thinking about the proof of soundness.

Due to the introduction of recursive types, λC_β is no long consistent so that not able to be used as a logic. But with the power of general recursion, the expressibility of λC_β is increased since more data types and functions can be mapped or encoded into λC_β . And more importantly, even with μ -notation, λC_β can still be proved to have the same properties as λC_β in the sense of decidability of type checking and soundness.

As what we previously illustrate in Section ??, the type checking of λC_{exp} can always terminate because the derivation is finite without the implicit conversion rule. With the μ -notation in λC_β , the decidability of type checking still holds because the type level recursion is explicitly controlled by cast operations. Notice that in the typing rule of cast^\uparrow and cast_\downarrow , the reduction is performed by one step. Thus the reduction sequences are always finite. Also by adopting the definitional equality, to judge if two terms are equal in the type checking is also decidable. Therefore, the new T_MU rule is decidable for type checking.

To prove the soundness, we only need to consider each one more case for subject reduction and progress, i.e. S_MU and T_MU. It is straightforward to verify these two rules still keeping the soundness.

6. Surface language

BRUNO: Jeremy, I think you should write up this section.

- Expand the core language with datatypes and pattern matching by encoding.
- Give translation rules.
- Encode GADTs and maybe other Haskell extensions? GADTs seems challenging, so perhaps some other examples would be datatypes like *Fixf*, and *Monad* as a record. Could formalize records in Haskell style.

In this section, we present the surface language (λC_{suf}) that supports simple datatypes and case analysis. Due to the expressiveness of λC_β , all these features can be elaborated into the core language without extending the built-in language constructs of λC_β . In what follows, we first give the syntax of λC_{suf} , followed by the extended typing rules, then we show the formal translation rules that translates λC_{suf} expressions into λC_β expressions. Finally we demonstrate the translation using a simple example.

6.1 Extended Syntax

The syntax of λC_{suf} is shown in Figure 7. Compared with λC_β , λC_{suf} has a new syntax category: a program, consisting of a list of datatype declarations, followed by an expression. An *algebraic data type* D is introduced as a top-level **data** declaration with its *data constructors*. The type of a data constructor K has the form:

$$K : \Pi \bar{u} : \bar{\kappa}^n. \Pi \bar{x} : \bar{\tau} \rightarrow D \bar{u}^n$$

The first n quantified type variables \bar{u} appear in the same order in the return type $D \bar{u}$. **Note that the use of Π to tie together the data constructor arguments makes it possible to let the types of some data constructor arguments depend on other data constructor arguments.** The **case** expression is conventional, used to break up values built with data constructors. The patterns of a case expression are flat (no nested patterns), and bind value variables.

Declarations		
<i>pgm</i>	$::= \overline{\text{decl}}; e$	Declarations
<i>decl</i>	$::= \mathbf{data} \ D \ \bar{u} : \bar{\kappa} = \overline{K \ \bar{\tau}}$	Datatype
Terms		
<i>u</i>	$::= x \mid K$	Variables and constructors
$e, \tau, \sigma, v, \kappa$	$::= u$	Term atoms
	\dots	
	$\mid \mathbf{case} \ e \ \mathbf{of} \ \overline{p \Rightarrow e}$	Case analysis
<i>p</i>	$::= K \ \bar{x} : \bar{\tau}$	Pattern
Environments		
Γ	$::= \emptyset$	Empty
	$\mid \Gamma, u : \tau$	Variable binding

Figure 7. Syntax of λC_{suf} (e for terms; τ, σ, v for types; κ for kinds)

With datatypes, it is easy to encode *records* as syntactic sugar of simple datatypes, as shown in Figure 8.

$$\begin{aligned} \mathbf{data} \ R \ \bar{u} : \bar{\kappa} &= K \ \{ \overline{S : \tau} \} \triangleq \\ \mathbf{data} \ R \ \bar{u} : \bar{\kappa} &= K \ \bar{\tau} \\ \mathbf{let} \ S_i : \Pi \bar{u} : \bar{\kappa}. R \ \bar{u} \rightarrow \tau_i &= \\ \lambda(u : \bar{\kappa}). \lambda l : R \ \bar{u}. \mathbf{case} \ l \ \mathbf{of} \ K \ \bar{x} : \bar{\tau} \Rightarrow x_i & \\ \mathbf{in} & \end{aligned}$$

Figure 8. Syntactic sugar for records

6.2 Extended Typing Rules

The type system of λC_{suf} is shown in Figure 9. To save space, we only show the new typing rules. Furthermore, we sometimes adopt the following syntactic convention:

$$\bar{\tau}^n \rightarrow \tau_r \equiv \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau_r$$

Rule (Pgm) type-checks a whole problem. It first type-checks the declarations, which in return gives a new typing environment. Combined with the original environment, it then checks the expression and return the result type. Rule (Data) type-checks datatype declarations by ensuring the well-formedness of the kinds of type constructors and the types of data constructors. Finally rule (Alt) validates the patterns by looking up the the existence of corresponding data constructors in the typing environment, replacing universal quantified type variables with proper concrete types.

$\Gamma \vdash \text{pgm} : \tau$	
(Pgm)	$\frac{\Gamma_0 \vdash \text{decl} : \Gamma_d \quad \Gamma = \Gamma_0, \Gamma_d \quad \Gamma \vdash e : \tau}{\Gamma_0 \vdash \text{decl}; e : \tau}$
$\Gamma \vdash \text{decl} : \Gamma_d$	
(Data)	$\frac{\Gamma \vdash \bar{\kappa} \rightarrow \star : \square \quad \Gamma, D : \bar{\kappa} \rightarrow \star, \bar{u} : \bar{\kappa} \vdash \bar{\tau} \rightarrow D \bar{u} : \star}{\Gamma \vdash (\text{data } D \bar{u} : \bar{\kappa} = \mid \bar{K} \bar{\tau}) : (D : \bar{\kappa} \rightarrow \star, \bar{K} : \Pi \bar{u} : \bar{\kappa}. \bar{\tau} \rightarrow D \bar{u})}$
$\Gamma \vdash e : \tau$	
(Case)	$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash_p p \Rightarrow e_2 : \sigma \rightarrow \tau}{\Gamma \vdash \text{case } e_1 \text{ of } \bar{p} \Rightarrow e_2 : \tau}$
$\Gamma \vdash_p p \Rightarrow e : \sigma \rightarrow \tau$	
(Alt)	$\frac{\theta = [\bar{u} := \bar{v}] \quad K : \Pi \bar{u} : \bar{\kappa}. \bar{\sigma} \rightarrow D \bar{u} \in \Gamma \quad \Gamma, x : \theta(\sigma) \vdash e : \tau}{\Gamma \vdash_p K x : \theta(\sigma) \Rightarrow e : D \bar{v} \rightarrow \tau}$

Figure 9. Typing rules of λC_{suf}

6.3 Translation Overview

We use a type-directed translation. The typing relations have the form:

$$\Gamma \vdash e : \tau \rightsquigarrow E$$

It states that λC_{β} expression E is the translation of λC_{suf} expression e of type τ . Figure 10 shows the translation rules, which are the typing rules in Figure 9 extended with the resulting expression E . In the translation, We require that applications of constructors to be *saturated*.

Among others, Rules (Case), (Alt) and (Data) are of the essence for the translation. Rule (Case) translates case expressions into applications by first type-converting the scrutinee expression, then applying it to the result type and a λC_{β} expression. Rule (Alt) translate each pattern into a lambda expression, with each variable in the pattern corresponding to a variable in the lambda expression in the same order. The body in the alternative is recursively translated and taken as the lambda body.

Rule (Data) does the most heavy work and deserves further explanation. First of all, it results in a incomplete expression (as can be seen by the incomplete *let* expressions), The result expression is supposed to be prepended to the translation of the last expression to form a complete λC_{β} expression, as specified by Rule (Pgm). Furthermore, each type constructor is translated as a lambda expression, with a recursive type as the body. Each data constructor is also translated as a lambda expression. Notice that we use cast operation in the lambda body to restore to the corresponding data-type.

The rest of the translation rules hold few surprises.

7. Related Work

8. Conclusion

Conclusion and related work.

Acknowledgments

Thanks to Blah. This work is supported by Blah.

References

- [1] T. Altenkirch, N. A. Danielsson, A. Löb, and N. Oury. $\Pi\Sigma$: Dependent types without the sugar. In *Functional and Logic Programming*, pages 40–55. Springer, 2010.
- [2] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

- [3] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. *ACM SIGPLAN Notices*, 49(1):33–45, 2014.
- [4] T. Coquand. *Une théorie des constructions*. PhD thesis, 1985.
- [5] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, Feb. 1988. ISSN 0890-5401. . URL [http://dx.doi.org/10.1016/0890-5401\(88\)90005-3](http://dx.doi.org/10.1016/0890-5401(88)90005-3).
- [6] S. P. Jones and E. Meijer. Henk: a typed intermediate language. 1997.
- [7] A. Middelkoop, A. Dijkstra, and S. D. Swierstra. A lean specification for gadts: system f with first-class equality proofs. *Higher-Order and Symbolic Computation*, 23(2):145–166, 2010.
- [8] J.-W. Roorde and J. Jeuring. Pure type systems for functional programming. 2007.
- [9] P. G. Severi and F.-J. J. de Vries. Pure type systems with corecursion on streams: from finite to infinitary normalisation. In *ACM SIGPLAN Notices*, volume 47, pages 141–152. ACM, 2012.
- [10] V. Sjöberg. *A Dependently Typed Language with Nontermination*. PhD thesis, University of Pennsylvania, 2015.
- [11] V. Sjöberg and S. Weirich. Programming up to congruence. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, pages 369–382, New York, NY, USA, 2015. ACM. .
- [12] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.
- [13] J. C. Vanderwaart, D. Dreyer, L. Petersen, K. Crary, R. Harper, and P. Cheng. *Typed compilation of recursive datatypes*, volume 38. ACM, 2003.
- [14] S. Weirich, J. Hsu, and R. A. Eisenberg. Towards dependently typed haskell: System fc with kind equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP*, volume 13. Citeseer, 2013.
- [15] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.

A. Full specification of source language

A.1 Syntax

See Figure 11.

A.2 Expression typing

See Figure 12.

	$\boxed{\Gamma \vdash e : \tau \rightsquigarrow E}$	
(Ax)	$\frac{}{\emptyset \vdash \star : \square \rightsquigarrow \star}$	
(Var)	$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow x}$	
(App)	$\frac{\Gamma \vdash e_1 : (\Pi x : \tau_2. \tau_1) \rightsquigarrow E_1 \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow E_2}{\Gamma \vdash e_1 e_2 : \tau_1[x := e_2] \rightsquigarrow E_1 E_2}$	
(Lam)	$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow E \quad \Gamma \vdash (\Pi x : \tau_1. \tau_2) : t}{\Gamma \vdash (\lambda x : \tau_1. e) : (\Pi x : \tau_1. \tau_2) \rightsquigarrow \lambda x : \tau_1. E}$	$t \in \{\star, \square\}$
(Pi)	$\frac{\Gamma \vdash \tau_1 : s \quad \Gamma, x : \tau_1 \vdash \tau_2 : t}{\Gamma \vdash (\Pi x : \tau_1. \tau_2) : t \rightsquigarrow \Pi x : \tau_1. \tau_2}$	$(s, t) \in \mathcal{R}$
(Mu)	$\frac{\Gamma, x : \tau \vdash e : \tau \rightsquigarrow E \quad \Gamma \vdash \tau : s}{\Gamma \vdash (\mu x : \tau. e) : \tau \rightsquigarrow \mu x : \tau. E}$	$s \in \{\star, \square\}$
(Fold)	$\frac{\Gamma \vdash e : \tau_2 \rightsquigarrow E \quad \Gamma \vdash \tau_1 : s \quad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\text{cast}^\uparrow_{[\tau_1]} e) : \tau_1 \rightsquigarrow \text{cast}^\uparrow_{[\tau_1]} E}$	
(Unfold)	$\frac{\Gamma \vdash e : \tau_1 \rightsquigarrow E \quad \Gamma \vdash \tau_2 : s \quad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\text{cast}_\downarrow e) : \tau_2 \rightsquigarrow \text{cast}_\downarrow E}$	
(Case)	$\frac{\Gamma \vdash e_1 : \sigma \rightsquigarrow E_1 \quad \Gamma \vdash_p p \Rightarrow e_2 : \sigma \rightarrow \tau \rightsquigarrow E_2}{\Gamma \vdash \text{case } e_1 \text{ of } \overline{p} \Rightarrow \overline{e_2} : \tau \rightsquigarrow (\text{cast}_\downarrow E_1) \tau \overline{E_2}}$	
	$\boxed{\Gamma \vdash_p p \Rightarrow e : \sigma \rightarrow \tau \rightsquigarrow E}$	
(Alt)	$\frac{\theta = [\overline{u} := \overline{v}]}{K : \Pi \overline{u} : \overline{\kappa}. \overline{\sigma} \rightarrow D \overline{u} \in \Gamma \quad \Gamma, x : \theta(\sigma) \vdash e : \tau \rightsquigarrow E}$ $\frac{}{\Gamma \vdash_p K x : \theta(\sigma) \Rightarrow e : D \overline{u} \rightarrow \tau \rightsquigarrow \lambda(x : \theta(\sigma)). E}$	
	$\boxed{\Gamma \vdash \text{decl} : \Gamma_d \rightsquigarrow E}$	
(Data)	$\frac{\Gamma \vdash \overline{\kappa} \rightarrow \star : \square \quad \Gamma, D : \overline{\kappa} \rightarrow \star, \overline{u} : \overline{\kappa} \vdash \overline{\tau} \rightarrow D \overline{u} : \star}{\Gamma \vdash (\text{data } D \overline{u} : \overline{\kappa} = \overline{K} \overline{\tau}) : (D : \overline{\kappa} \rightarrow \star, \overline{K} : \Pi \overline{u} : \overline{\kappa}. \overline{\tau} \rightarrow D \overline{u}) \rightsquigarrow E}$ $E ::= \text{let } D : \overline{\kappa} \rightarrow \star = \lambda \overline{u} : \overline{\kappa}. \mu X : \star. \Pi b : \star. (\overline{\tau}[D \overline{u} := X] \rightarrow b) \rightarrow b \text{ in}$ $\text{let } K_i : \Pi \overline{u} : \overline{\kappa}. \overline{\tau} \rightarrow D \overline{u} = \lambda(u : \kappa). \lambda(x : \tau).$ $\text{cast}^\uparrow_{[D \overline{u}]} (\lambda(b : \star)(c : \overline{\tau} \rightarrow b). c_i \overline{x}) \text{ in}$	
	$\boxed{\Gamma \vdash \text{pgm} : \tau \rightsquigarrow E}$	
(Pgm)	$\frac{\Gamma_0 \vdash \text{decl} : \Gamma_d \rightsquigarrow E_1 \quad \Gamma = \Gamma_0, \overline{\Gamma}_d \quad \Gamma \vdash e : \tau \rightsquigarrow E}{\Gamma_0 \vdash \text{decl}; e : \tau \rightsquigarrow E_1 \oplus E}$	

Figure 10. Type-directed translation from λC_{suf} to λC_β

A.3 Translation to the core

See Figure 13.

B. Proofs about core language

B.1 Properties

Lemma B.1 (Free variable lemma). *If $\Gamma \vdash e : \tau$, then $\text{FV}(e) \subseteq \text{dom}(\Gamma)$ and $\text{FV}(\tau) \subseteq \text{dom}(\Gamma)$.*

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. We only treat cases T_MU, T_CASTUP and T_CASTDOWN (since proofs of other cases are the same as λC [2]):

Case T_MU: From premises of $\Gamma \vdash (\mu x : \tau. e_1) : \tau$, by induction hypothesis, we have $\text{FV}(e_1) \subseteq \text{dom}(\Gamma) \cup \{x\}$ and $\text{FV}(\tau) \subseteq \text{dom}(\Gamma)$. Thus the result follows by $\text{FV}(\mu x : \tau. e_1) = \text{FV}(e_1) \setminus \{x\} \subseteq \text{dom}(\Gamma)$ and $\text{FV}(\tau) \subseteq \text{dom}(\Gamma)$.

Case T_CASTUP: Since $\text{FV}(\text{cast}^\uparrow_{[\tau]} e_1) = \text{FV}(e_1)$, the result follows directly by the induction hypothesis.

Case T_CASTDOWN: Since $\text{FV}(\text{cast}_\downarrow e_1) = \text{FV}(e_1)$, the result follows directly by the induction hypothesis. \square

Lemma B.2 (Substitution lemma). *If $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$ and $\Gamma_1 \vdash e_2 : \sigma$, then $\Gamma_1, \Gamma_2[x \mapsto e_2] \vdash e_1[x \mapsto e_2] : \tau[x \mapsto e_2]$.*

Proof. By induction on the derivation of $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$. Let $e^* \equiv e[x \mapsto e_2]$. Then the result can be written as $\Gamma_1, \Gamma_2^* \vdash e_1^* : \tau^*$. We only treat cases T_MU, T_CASTUP and T_CASTDOWN. Consider the last step of derivation of the following cases:

Case T_MU:
$$\frac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau \quad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau : \star}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\mu y : \tau. e_1) : \tau}$$

 By induction hypothesis, we have $\Gamma_1, \Gamma_2^* \vdash e_1^* : \tau^*$ and $\Gamma_1, \Gamma_2^* \vdash \tau^* : \star$. Then by the deviation rule, $\Gamma_1, \Gamma_2^* \vdash (\mu y : \tau^*. e_1^*) : \tau^*$. Thus we have $\Gamma_1, \Gamma_2^* \vdash (\mu y : \tau. e_1)^* : \tau^*$ which is just the result.

pgm	$::=$ $\mid \overline{decl}; E$	Program Declarations
$decl$	$::=$ $\mid \text{data } \overline{Du : \kappa} = \overline{KT}$	Declarations Datatype declaration
u, v	$::=$ $\mid x$ $\mid K$	Atoms Variable Data constructor
p	$::=$ $\mid Kx : \overline{T}$	Patterns Pattern
E, T, S, κ	$::=$ $\mid u$ $\mid \star$ $\mid E_1 E_2$ $\mid \lambda x : T. E$ $\mid \Pi x : T_1. T_2$ $\mid \mu x : T. E$ $\mid \text{case } E_1 \text{ of } p \Rightarrow \overline{E_2}$ $\mid \text{let } x : T = E_2 \text{ in } E_1$	Expressions Atom Type of types Application Abstraction Product General recursion Case analysis Let binding
Γ	$::=$ $\mid \emptyset$ $\mid \Gamma, u : T$	Contexts Empty Atom binding
$T_1 \rightarrow T_2$	$\triangleq \Pi x : T_1. T_2$	$x \notin \text{FV}(T_2)$ Syntactic Sugar Function type
$(x : T_1) \rightarrow T_2$	$\triangleq \Pi x : T_1. T_2$	$x \in \text{FV}(T_2)$ Dependent function type
$\text{data } \overline{Ru : \kappa} = K\{\overline{N : T}\}$	$\triangleq \text{data } \overline{Ru : \kappa} = \overline{KT};$ $\text{let } N_i : (\overline{u : \kappa}) \rightarrow \overline{Ru} \rightarrow T_i =$ $\lambda \overline{u : \kappa}. \lambda y : \overline{Ru}. \text{case } y \text{ of } Kx : \overline{T} \Rightarrow x_i \text{ in}$	$\forall i$ Record

Figure 11. Syntax of source language

Case T_CASTUP: $\frac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau_2 \quad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau_1 : \star \quad \tau_1 \longrightarrow \tau_2}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\text{cast}^\uparrow[\tau_1]e_1) : \tau_1}$
 By induction hypothesis, we have $\Gamma_1, \Gamma_2^* \vdash e_1^* : \tau_2^*$, $\Gamma_1, \Gamma_2^* \vdash \tau_1^* : \star$ and $\tau_1 \longrightarrow \tau_2$. By the definition of substitution, we can obtain $\tau_1^* \longrightarrow \tau_2^*$ by $\tau_1 \longrightarrow \tau_2$. Then by the deviation rule, $\Gamma_1, \Gamma_2^* \vdash (\text{cast}^\uparrow[\tau_1^*]e_1^*) : \tau_1^*$. Thus we have $\Gamma_1, \Gamma_2^* \vdash (\text{cast}^\uparrow[\tau_1]e_1)^* : \tau_1^*$ which is just the result.

Case T_CASTDOWN: $\frac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau_1 \quad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau_2 : \star \quad \tau_1 \longrightarrow \tau_2}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\text{cast}_\downarrow e_1) : \tau_2}$
 By induction hypothesis, we have $\Gamma_1, \Gamma_2^* \vdash e_1^* : \tau_1^*$, $\Gamma_1, \Gamma_2^* \vdash \tau_2^* : \star$ and $\tau_1 \longrightarrow \tau_2$ thus $\tau_1^* \longrightarrow \tau_2^*$. Then by the deviation rule, $\Gamma_1, \Gamma_2^* \vdash (\text{cast}_\downarrow e_1^*) : \tau_2^*$. Thus we have $\Gamma_1, \Gamma_2^* \vdash (\text{cast}_\downarrow e_1)^* : \tau_2^*$ which is just the result.

□

Lemma B.3 (Generation lemma).

- (1) If $\Gamma \vdash x : \sigma$, then there exist an expression τ such that $\tau \equiv \sigma$, $\Gamma \vdash \tau : \star$ and $x : \tau \in \Gamma$.
- (2) If $\Gamma \vdash e_1 e_2 : \sigma$, then there exist expressions τ_1 and τ_2 such that $\Gamma \vdash e_1 : (\Pi x : \tau_1. \tau_2)$, $\Gamma \vdash e_2 : \tau_1$ and $\sigma \equiv \tau_2[x \mapsto e_2]$.

- (3) If $\Gamma \vdash (\lambda x : \tau_1. e) : \sigma$, then there exist an expression τ_2 such that $\sigma \equiv \Pi x : \tau_1. \tau_2$ where $\Gamma \vdash (\Pi x : \tau_1. \tau_2) : \star$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$.
- (4) If $\Gamma \vdash (\Pi x : \tau_1. \tau_2) : \sigma$, then $\sigma \equiv \star$, $\Gamma \vdash \tau_1 : \star$ and $\Gamma, x : \tau_1 \vdash \tau_2 : \star$.
- (5) If $\Gamma \vdash (\mu x : \tau. e) : \sigma$, then $\Gamma \vdash \tau : \star$, $\sigma \equiv \tau$ and $\Gamma, x : \tau \vdash e : \tau$.
- (6) If $\Gamma \vdash (\text{cast}^\uparrow[\tau_1]e) : \sigma$, then there exist an expression τ_2 such that $\Gamma \vdash e : \tau_2$, $\Gamma \vdash \tau_1 : \star$, $\tau_1 \longrightarrow \tau_2$ and $\sigma \equiv \tau_1$.
- (7) If $\Gamma \vdash (\text{cast}_\downarrow e) : \sigma$, then there exist expressions τ_1, τ_2 such that $\Gamma \vdash e : \tau_1$, $\Gamma \vdash \tau_2 : \star$, $\tau_1 \longrightarrow \tau_2$ and $\sigma \equiv \tau_2$.

Proof. Consider a derivation of $\Gamma \vdash e : \sigma$ for one of cases in the lemma. Note that rule T.WEAK does not change e , then we can follow the process of derivation until expression e is introduced the first time. The last step of derivation can be done by

- rule T.VAR for case 1;
- rule T.APP for case 2;
- rule T.LAM for case 3;
- rule T.PI for case 4;
- rule T.MU for case 5;
- rule T.CASTUP for case 6;
- rule T.CASTDOWN for case 7.

$\boxed{\vdash \Gamma}$ Well-formed context

$$\frac{}{\vdash \emptyset} \text{CTX_EMPTY}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash T : \star}{\vdash \Gamma, x : T} \text{CTX_VAR}$$

$\boxed{\Gamma \vdash \text{pgm} : T}$ Program context

$$\frac{\Gamma_0 \vdash \text{decl} : \Gamma' \quad \Gamma = \Gamma_0, \Gamma' \quad \Gamma \vdash E : T}{\Gamma_0 \vdash (\text{decl}; E) : T} \text{TPGM_PGM}$$

$\boxed{\Gamma \vdash \text{decl} : \Gamma'}$ Datatype declaration

$$\frac{\Gamma \vdash (\bar{u} : \bar{\kappa}) \rightarrow \star : \star \quad \overline{\Gamma, D : (\bar{u} : \bar{\kappa}) \rightarrow \star, \bar{u} : \bar{\kappa} \vdash \bar{T} \rightarrow D\bar{u} : \star}}{\Gamma \vdash (\text{data } D\bar{u} : \bar{\kappa} = \overline{K\bar{T}}) : (D : (\bar{u} : \bar{\kappa}) \rightarrow \star, K : (\bar{u} : \bar{\kappa}) \rightarrow \bar{T} \rightarrow D\bar{u})} \text{TDECL_DATA}$$

$\boxed{\Gamma \vdash p \Rightarrow E : S \rightarrow T}$ Pattern typing

$$\frac{K : (\bar{u} : \bar{\kappa}) \rightarrow \bar{S} \rightarrow D\bar{u} \in \Gamma \quad \Gamma, x : S[\bar{u} \mapsto \bar{v}] \vdash E : T \quad \Gamma \vdash S[\bar{u} \mapsto \bar{v}] : \star}{\Gamma \vdash Kx : S[\bar{u} \mapsto \bar{v}] \Rightarrow E : D\bar{v} \rightarrow T} \text{TPAT_ALT}$$

$\boxed{\Gamma \vdash E : T}$ Expression typing

$$\frac{}{\emptyset \vdash \star : \star} \text{TS_AX}$$

$$\frac{\vdash \Gamma \quad x : T \in \Gamma}{\Gamma \vdash x : T} \text{TS_VAR}$$

$$\frac{\Gamma \vdash E_1 : (\Pi x : T_2.T_1) \quad \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1 E_2 : T_1[x \mapsto E_2]} \text{TS_APP}$$

$$\frac{\Gamma, x : T_1 \vdash E : T_2 \quad \Gamma \vdash (\Pi x : T_1.T_2) : \star}{\Gamma \vdash (\lambda x : T_1.E) : (\Pi x : T_1.T_2)} \text{TS_LAM}$$

$$\frac{\Gamma \vdash T_1 : \star \quad \Gamma, x : T_1 \vdash T_2 : \star}{\Gamma \vdash (\Pi x : T_1.T_2) : \star} \text{TS_PI}$$

$$\frac{\Gamma, x : T \vdash E : T \quad \Gamma \vdash T : \star}{\Gamma \vdash (\mu x : T.E) : T} \text{TS_MU}$$

$$\frac{\Gamma \vdash E_1 : S \quad \overline{\Gamma \vdash p \Rightarrow E_2 : S \rightarrow T} \quad \overline{\Gamma \vdash S \rightarrow T : \star}}{\Gamma \vdash \text{case } E_1 \text{ of } \overline{p \Rightarrow E_2} : T} \text{TS_CASE}$$

$$\frac{\Gamma \vdash E_2 : T \quad \Gamma \vdash E_1[x \mapsto E_2] : S}{\Gamma \vdash \text{let } x : T = E_2 \text{ in } E_1 : S} \text{TS_LET}$$

Figure 12. Typing rules of source language

In each case, assume the conclusion of the rule is $\Gamma' \vdash e : \tau'$ where $\Gamma' \subseteq \Gamma$ and $\tau' \equiv \sigma$. Then by inspection of used derivation rules, it can be shown that the statement of the lemma holds and is the only possible case. \square

Lemma B.4 (Correctness of types). *If $\Gamma \vdash e : \tau$ then $\tau \equiv \star$ or $\Gamma \vdash \tau : \star$.*

Proof. Trivial induction on the derivation of $\Gamma \vdash e : \tau$ using Lemma B.3. \square

B.2 Decidability of type checking

Lemma B.5 (Uniqueness of one-step reduction). *The relation \longrightarrow , i.e. one-step reduction, is **unique** in the sense that given e there is at most one e' such that $e \longrightarrow e'$.*

Proof. By induction on the structure of e :

Case $e = v$: e has one of the following forms: (1) \star , (2) x , (3) $\lambda x : \tau.e$, (4) $\Pi x : \tau_1.\tau_2$, (5) $\text{cast}^\uparrow[\tau]e$, which cannot match any rules of \longrightarrow . Thus there is no e' such that $e \longrightarrow e'$.
Case $e = (\lambda x : \tau.e_1) e_2$: There is a unique $e' = e_1[x \mapsto e_2]$ by rule S_BETA.

$\boxed{\Gamma \vdash \text{pgm} : T \rightsquigarrow e}$	Program translation
$\frac{\overline{\Gamma_0 \vdash \text{decl} : \Gamma' \rightsquigarrow e_1} \quad \Gamma = \Gamma_0, \overline{\Gamma'} \quad \Gamma \vdash E : T \rightsquigarrow e}{\Gamma_0 \vdash (\overline{\text{decl}}; E) : T \rightsquigarrow \overline{e_1} \uplus e} \quad \text{TRPGM_PGM}$	
$\boxed{\Gamma \vdash \text{decl} : \Gamma' \rightsquigarrow e}$	Datatype translation
$\frac{\Gamma \vdash (\overline{u : \kappa}) \rightarrow \star : \star \rightsquigarrow (\overline{u : \sigma}) \rightarrow \star \quad \overline{\Gamma, D : (\overline{u : \kappa}) \rightarrow \star, \overline{u : \kappa} \vdash \overline{T} \rightarrow D\overline{u} : \star \rightsquigarrow \overline{\tau} \rightarrow D\overline{u}}}{\Gamma \vdash (\text{data } D\overline{u} : \overline{\kappa} = K\overline{T}) : (D : (\overline{u : \kappa}) \rightarrow \star, K : (\overline{u : \kappa}) \rightarrow \overline{T} \rightarrow D\overline{u}) \rightsquigarrow e} \quad \text{TRDECL_DATA}$ $e \triangleq \text{let } D : (\overline{u : \sigma}) \rightarrow \star = \mu X : (\overline{u : \sigma}) \rightarrow \star. \lambda \overline{u} : \overline{\sigma}. (\alpha : \star) \rightarrow \overline{(\tau[D \mapsto X] \rightarrow \alpha)} \rightarrow \alpha \text{ in}$ $\text{let } K_i : (\overline{u : \sigma}) \rightarrow \overline{\tau} \rightarrow D\overline{u} = \lambda \overline{u} : \overline{\sigma}. \lambda \overline{x} : \overline{\tau}. \text{cast}^n_{\downarrow} [D\overline{u}] (\lambda \alpha : \star. \lambda \overline{b} : \overline{\tau} \rightarrow \alpha. b_i \overline{x}) \text{ in}$	
$\boxed{\Gamma \vdash p \Rightarrow E : S \rightarrow T \rightsquigarrow e}$	Pattern translation
$\frac{K : (\overline{u : \kappa}) \rightarrow \overline{S} \rightarrow D\overline{u} \in \Gamma \quad \Gamma, x : \overline{S[\overline{u} \mapsto \overline{v}]} \vdash E : T \rightsquigarrow e \quad \Gamma \vdash S[\overline{u} \mapsto \overline{v}] : \star \rightsquigarrow \sigma}{\Gamma \vdash Kx : S[\overline{u} \mapsto \overline{v}] \Rightarrow E : D\overline{v} \rightarrow T \rightsquigarrow \lambda \overline{x} : \overline{\sigma}. e} \quad \text{TRPAT_ALT}$	
$\boxed{\Gamma \vdash E : T \rightsquigarrow e}$	Expression translation
$\frac{}{\emptyset \vdash \star : \star \rightsquigarrow \star} \quad \text{TR_AX}$ $\frac{\vdash \Gamma \quad x : T \in \Gamma}{\Gamma \vdash x : T \rightsquigarrow x} \quad \text{TR_VAR}$ $\frac{\Gamma \vdash E_1 : (\Pi x : T_2. T_1) \rightsquigarrow e_1 \quad \Gamma \vdash E_2 : T_2 \rightsquigarrow e_2}{\Gamma \vdash E_1 E_2 : T_1[x \mapsto E_2] \rightsquigarrow e_1 e_2} \quad \text{TR_APP}$ $\frac{\Gamma, x : T_1 \vdash E : T_2 \rightsquigarrow e \quad \Gamma \vdash (\Pi x : T_1. T_2) : \star \rightsquigarrow \Pi x : \tau_1. \tau_2}{\Gamma \vdash (\lambda x : T_1. E) : (\Pi x : T_1. T_2) \rightsquigarrow \lambda x : \tau_1. e} \quad \text{TR_LAM}$ $\frac{\Gamma \vdash T_1 : \star \rightsquigarrow \tau_1 \quad \Gamma, x : T_1 \vdash T_2 : \star \rightsquigarrow \tau_2}{\Gamma \vdash (\Pi x : T_1. T_2) : \star \rightsquigarrow \Pi x : \tau_1. \tau_2} \quad \text{TR_PI}$ $\frac{\Gamma, x : T \vdash E : T \rightsquigarrow e \quad \Gamma \vdash T : \star \rightsquigarrow \tau}{\Gamma \vdash (\mu x : T. E) : T \rightsquigarrow \mu x : \tau. e} \quad \text{TR_MU}$ $\frac{\Gamma \vdash E_1 : S \rightsquigarrow e_1 \quad \overline{\Gamma \vdash p \Rightarrow E_2 : S \rightarrow T \rightsquigarrow e_2} \quad \overline{\Gamma \vdash S \rightarrow T : \star \rightsquigarrow \sigma \rightarrow \tau}}{\Gamma \vdash \text{case } E_1 \text{ of } \overline{p \Rightarrow E_2} : T \rightsquigarrow (\text{cast}^n_{\downarrow} e_1) \tau \overline{e_2}} \quad \text{TR_CASE}$ $\frac{\Gamma \vdash E_2 : T \rightsquigarrow e_2 \quad \Gamma \vdash E_1[x \mapsto E_2] : S \rightsquigarrow e_1[x \mapsto e_2]}{\Gamma \vdash \text{let } x : T = E_2 \text{ in } E_1 : S \rightsquigarrow e_1[x \mapsto e_2]} \quad \text{TR_LET}$	

Figure 13. Translation rules of source language

Case $e = \text{cast}_{\downarrow}(\text{cast}^{\uparrow}[\tau]e)$: There is a unique $e' = e$ by rule S_CASTDOWNUP.

Case $e = \mu x : \tau. e$: There is a unique $e' = e[x \mapsto \mu x : \tau. e]$ by rule S_MU.

Case $e = e_1 e_2$ **and** e_1 **is not a λ -term**: If $e_1 = v$, there is no e'_1 such that $e_1 \longrightarrow e'_1$. Since e_1 is not a λ -term, there is no rule to reduce e . Thus there is no e' such that $e \longrightarrow e'$. Otherwise, there exists some e'_1 such that $e_1 \longrightarrow e'_1$. By the induction hypothesis, e'_1 is unique reduction of e_1 . Thus by rule S_APP, $e' = e'_1 e_2$ is the unique reduction for e .

Case $e = \text{cast}_{\downarrow} e_1$ **and** e_1 **is not a cast^{\uparrow} -term**: If $e_1 = v$, there is no e'_1 such that $e_1 \longrightarrow e'_1$. Since e_1 is not a cast^{\uparrow} -term, there is no rule to reduce e . Thus there is no e' such that $e \longrightarrow e'$.

Otherwise, there exists some e'_1 such that $e_1 \longrightarrow e'_1$. By the induction hypothesis, e'_1 is unique reduction of e_1 . Thus by rule S_CASTDOWN, $e' = \text{cast}_{\downarrow} e'_1$ is the unique reduction for e .

□

Lemma B.6 (Decidability of type checking). *There is a decidable algorithm which given Γ, e computes the unique τ such that $\Gamma \vdash e : \tau$ or reports there is no such τ .*

Proof. By induction on the structure of e :

Case $e = \star$: Trivial by applying T_AX and $\tau \equiv \star$.

Case $e = x$: By Lemma ??, we only need to consider context Γ that is well-formed. By rule TS_VAR, if $x : \tau \in \Gamma$, τ is the unique type of x .

Case $e = e_1 e_2$, or $\lambda x : \tau_1. e_1$, or $\Pi x : \tau_1. \tau_2$, or $\mu x : \tau. e_1$: Trivial according to Lemma B.3 by using rule T_APP, T_LAM, T_PI, or T_MU respectively.

Case $e = \text{cast}^\uparrow[\tau_1] e_1$: From the premises of rule T_CASTUP, by induction hypothesis, we can derive the type of e_1 as τ_2 , and check whether τ_1 is legal, i.e. its sorts is \star . If τ_1 is legal, by Lemma B.5, there is at most one τ'_1 such that $\tau_1 \longrightarrow \tau'_1$. If such τ'_1 does not exist, then we report the type checking is failed. Otherwise, we examine if τ'_1 is syntactically equal to τ_2 , i.e. $\tau'_1 \equiv \tau_2$. If the equality holds, we obtain the unique type of e which is τ_1 . Otherwise, we report e fails to type check.

Case $e = \text{cast}_\downarrow e_1$: From the premises of rule T_CASTDOWN, by induction hypothesis, we can derive the type of e_1 as τ_1 . By Lemma B.5, there is at most one τ_2 such that $\tau_1 \longrightarrow \tau_2$. If such τ_2 exists and its sorts is \star , we have found the unique type of e is τ_2 . Otherwise, we report e fails to type check.

□

B.3 Soundness

Definition B.7 (Multi-step reduction). *The relation \rightarrow is the transitive and reflexive closure of \longrightarrow .*

Lemma B.8 (Subject reduction). *If $\Gamma \vdash e : \sigma$ and $e \rightarrow e'$ then $\Gamma \vdash e' : \sigma$.*

Proof. We prove the case for one-step reduction, i.e. $e \longrightarrow e'$. The lemma can follow by induction on the number of one-step reductions of $e \rightarrow e'$. The proof is by induction with respect to the definition of one-step reduction \longrightarrow as follows:

Case $\frac{(\lambda x : \tau. e_1) e_2 \longrightarrow e_1[x \mapsto e_2]}{\text{S_BETA:}}$

Suppose $\Gamma \vdash (\lambda x : \tau_1. e_1) e_2 : \sigma$ and $\Gamma \vdash e_1[x \mapsto e_2] : \sigma'$. By Lemma B.3(2), there exist expressions τ'_1 and τ_2 such that

$$\begin{aligned} \Gamma \vdash (\lambda x : \tau_1. e_1) : (\Pi x : \tau'_1. \tau_2) \quad (1) \\ \Gamma \vdash e_2 : \tau'_1 \\ \sigma \equiv \tau_2[x \mapsto e_2] \end{aligned}$$

By Lemma B.3(3), the judgement (1) implies that there exists an expression τ'_2 such that

$$\begin{aligned} \Pi x : \tau'_1. \tau_2 \equiv \Pi x : \tau_1. \tau'_2 \quad (2) \\ \Gamma, x : \tau_1 \vdash e_1 : \tau'_2 \end{aligned}$$

Hence, by (2) we have $\tau_1 \equiv \tau'_1$ and $\tau_2 \equiv \tau'_2$. Then we can obtain $\Gamma, x : \tau_1 \vdash e_1 : \tau_2$ and $\Gamma \vdash e_2 : \tau_1$. By Lemma B.2, we have $\Gamma \vdash e_1[x \mapsto e_2] : \tau_2[x \mapsto e_2]$. Therefore, we conclude with $\sigma' \equiv \tau_2[x \mapsto e_2] \equiv \sigma$.

Case $\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \text{S_APP:}$

Suppose $\Gamma \vdash e_1 e_2 : \sigma$ and $\Gamma \vdash e'_1 e_2 : \sigma'$. By Lemma B.3(2), there exist expressions τ_1 and τ_2 such that

$$\begin{aligned} \Gamma \vdash e_1 : (\Pi x : \tau_1. \tau_2) \\ \Gamma \vdash e_2 : \tau_1 \\ \sigma \equiv \tau_2[x \mapsto e_2] \end{aligned}$$

By induction hypothesis, we have $\Gamma \vdash e'_1 : (\Pi x : \tau_1. \tau_2)$. By rule T_APP, we obtain $\Gamma \vdash e'_1 e_2 : \tau_2[x \mapsto e_2]$. Therefore, $\sigma' \equiv \tau_2[x \mapsto e_2] \equiv \sigma$.

Case $\frac{e \longrightarrow e'}{\text{cast}_\downarrow e \longrightarrow \text{cast}_\downarrow e'} \text{S_CASTDOWN:}$

Suppose $\Gamma \vdash \text{cast}_\downarrow e : \sigma$ and $\Gamma \vdash \text{cast}_\downarrow e' : \sigma'$. By Lemma B.3(7), there exist expressions τ_1, τ_2 such that

$$\begin{aligned} \Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_2 : \star \\ \tau_1 \longrightarrow \tau_2 \quad \sigma \equiv \tau_2 \end{aligned}$$

By induction hypothesis, we have $\Gamma \vdash e' : \tau_1$. By rule T_CASTDOWN, we obtain $\Gamma \vdash \text{cast}_\downarrow e' : \tau_2$. Therefore, $\sigma' \equiv \tau_2 \equiv \sigma$.

Case $\frac{}{\text{cast}_\downarrow (\text{cast}^\uparrow[\tau] e) \longrightarrow e} \text{S_CASTDOWNUP:}$

Suppose $\Gamma \vdash \text{cast}_\downarrow (\text{cast}^\uparrow[\tau_1] e) : \sigma$ and $\Gamma \vdash e : \sigma'$. By Lemma B.3(7), there exist expressions τ'_1, τ_2 such that

$$\Gamma \vdash (\text{cast}^\uparrow[\tau_1] e) : \tau'_1 \quad (3)$$

$$\tau'_1 \longrightarrow \tau_2 \quad (4)$$

$$\sigma \equiv \tau_2 \quad (5)$$

By Lemma B.3(6), the judgement (3) implies that there exists an expression τ'_2 such that

$$\Gamma \vdash e : \tau'_2 \quad (6)$$

$$\tau_1 \longrightarrow \tau'_2 \quad (7)$$

$$\tau'_1 \equiv \tau_1 \quad (8)$$

By (4, 7, 8) and Lemma B.5 we obtain $\tau_2 \equiv \tau'_2$. From (6) we have $\sigma' \equiv \tau'_2$. Therefore, by (5), $\sigma' \equiv \tau'_2 \equiv \tau_2 \equiv \sigma$.

Case $\frac{\mu x : \tau. e \longrightarrow e[x \mapsto \mu x : \tau. e]}{\text{S_MU:}}$

Suppose $\Gamma \vdash (\mu x : \tau. e) : \sigma$ and $\Gamma \vdash e[x \mapsto \mu x : \tau. e] : \sigma'$. By Lemma B.3(5), we have $\sigma \equiv \tau$ and $\Gamma, x : \tau \vdash e : \tau$. Then we obtain $\Gamma \vdash (\mu x : \tau. e) : \tau$. Thus by Lemma B.2, we have $\Gamma \vdash e[x \mapsto \mu x : \tau. e] : \tau[x \mapsto \mu x : \tau. e]$.

Note that $x : \tau$, i.e. the type of x is τ , then $x \notin \text{FV}(\tau)$ holds implicitly. Hence, by the definition of substitution, we obtain $\tau[x \mapsto \mu x : \tau. e] \equiv \tau$. Therefore, $\sigma' \equiv \tau[x \mapsto \mu x : \tau. e] \equiv \tau \equiv \sigma$.

□

Lemma B.9 (Progress). *If $\vdash e : \sigma$ then either e is a value v or there exists e' such that $e \longrightarrow e'$.*

Proof. By induction on the derivation of $\vdash e : \sigma$ as follows:

Case $e = v$: Trivial, since e is already a value that has one of the following forms: (1) \star , (2) x , (3) $\lambda x : \tau. e$, (4) $\Pi x : \tau_1. \tau_2$, (5) $\text{cast}^\uparrow[\tau] e$.

Case $e = e_1 e_2$: By Lemma B.3(2), there exist expressions τ_1 and τ_2 such that $\vdash e_1 : (\Pi x : \tau_1. \tau_2)$ and $\vdash e_2 : \tau_1$. Consider whether e_1 is a value:

- If $e_1 = v$, by Lemma B.3(3), it must be a λ -term such that $e_1 \equiv \lambda x : \tau_1. e'_1$ for some e'_1 satisfying $\vdash e'_1 : \tau_2$. Then by rule S_BETA, we have $(\lambda x : \tau_1. e'_1) e_2 \longrightarrow e'_1[x \mapsto e_2]$. Thus, there exists $e' \equiv e'_1[x \mapsto e_2]$ such that $e \longrightarrow e'$.
- Otherwise, by induction hypothesis, there exists e'_1 such that $e_1 \longrightarrow e'_1$. Then by rule S_APP, we have $e_1 e_2 \longrightarrow e'_1 e_2$. Thus, there exists $e' \equiv e'_1 e_2$ such that $e \longrightarrow e'$.

Case $e = \text{cast}_\downarrow e_1$: By Lemma B.3(7), there exist expressions τ_1 and τ_2 such that $\vdash e_1 : \tau_1$ and $\tau_1 \longrightarrow \tau_2$. Consider whether e_1 is a value:

- If $e_1 = v$, by Lemma B.3(6), it must be a cast^\uparrow -term such that $e_1 \equiv \text{cast}^\uparrow[\tau_1] e'_1$ for some e'_1 satisfying $\vdash e'_1 : \tau_2$. Then by rule S_CASTDOWNUP, we can obtain $\text{cast}_\downarrow (\text{cast}^\uparrow[\tau_1] e'_1) \longrightarrow e'_1$. Thus, there exists $e' \equiv e'_1$ such that $e \longrightarrow e'$.

- Otherwise, by induction hypothesis, there exists e'_1 such that $e_1 \longrightarrow e'_1$. Then by rule S_CASTDOWN, we have $\text{cast}_\downarrow e_1 \longrightarrow \text{cast}_\downarrow e'_1$. Thus, there exists $e' \equiv \text{cast}_\downarrow e'_1$ such that $e \longrightarrow e'$.

Case $e = \mu x : \tau. e_1$: By rule S_MU, there always exists $e' \equiv e_1[x \mapsto \mu x : \tau. e_1]$.

□