

# Type-Level Computation One Step at a Time

Or: Decidable Type-Checking in the presence of Type-Level General Recursion

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 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

*These are definitely drafts and only some main points are listed in each section.*

a) Motivations:

- Because of the reluctance to introduce dependent types<sup>1</sup>, 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  $\mu$  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  $\beta$ -reduction for congruence closure [11].
- $\Pi\Sigma$  [1] is a simple, dependently-typed core language for expressing high-level constructions<sup>2</sup>. 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

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

<sup>2</sup> But the paper didn't give any meta-theories about the language.

form of dependent types into the system<sup>3</sup>, which mixes up types and kinds.

## 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  $\lambda C_\beta$ . 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 §4.

### 2.1 Calculus of Constructions

$\lambda C_\beta$  is based on the *Calculus of Constructions* ( $\lambda C$ ) [5], which is a higher-order typed lambda calculus. One “unconventional” feature of  $\lambda C$  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  $\beta$ -equality of  $\tau_1$  and  $\tau_2$ . Note that in  $\lambda C$ , 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  $\text{Int}$  and

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

Without the conversion rule,  $f$  cannot be applied to, say  $3$  in  $\lambda C$ . Given that  $f$  is actually  $\beta$ -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  $\lambda C$ ,  $\lambda C_\beta$  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:  $\text{cast}^\uparrow$  and  $\text{cast}_\downarrow$ .

In order to have a better intuition, let us consider the same example from §2.1. In  $\lambda C_\beta$ ,  $f\ 3$  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,  $\text{cast}^\uparrow$  is actually doing type conversion since the type of  $3$  is  $\text{Int}$  and  $(\lambda y : \star.y) \text{Int}$  can be reduced to  $\text{Int}$ .

The dual operation of  $\text{cast}^\uparrow$  is  $\text{cast}_\downarrow$ . The use of  $\text{cast}_\downarrow$  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  $\text{cast}_\downarrow$  reduces the type of  $z$  to  $\text{Int}$ .

### 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  $\lambda C$  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  $\lambda C_\beta$  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  $\text{loop}$  stands for any diverging computation and of type  $\text{Int}$ . What would happen if we try to type check the following application:

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

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

This is not the case for  $\lambda C_\beta$ : (i) it has no such conversion rule, therefore the type checker would do syntactic comparison between the two terms instead of  $\beta$ -equality in the above example; and (ii) one would need to write infinite number of  $\text{cast}_\downarrow$ ’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,  $\lambda C_\beta$  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  $\lambda C_\beta$  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.  $\lambda C_\beta$  differs from other programming languages in that it unifies both recursion and recursive types by the same  $\mu$  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*.  $\lambda C_\beta$  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  $\lambda C_\beta$ , this is witnessed by first  $\text{cast}^\uparrow$ , then  $\text{cast}_\downarrow$ . 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:

$$\text{cast}_\downarrow z : \text{Int} \rightarrow H$$

$$\text{cast}_\downarrow(\text{cast}_\downarrow z) : \text{Int} \rightarrow \text{Int} \rightarrow H$$

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

*Recursion.* The same  $\mu$  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  $\mu$  primitive:

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

which is exactly doing recursive unfolding of the same term.

<sup>3</sup>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>.

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  $\mu$  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  $\mu$  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:

```
data Nat = Z | S 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:

```
let Z : Nat = cast† [Nat] ( $\lambda B : \star. \lambda z : B. \lambda f : Nat \rightarrow B. z$ )
in
let S : Nat  $\rightarrow$  Nat =  $\lambda n : Nat.$ 
  cast† [Nat] ( $\lambda B : \star. \lambda z : B. \lambda f : Nat \rightarrow B. f\ n$ )
in
```

Thanks to the explicit type conversion rules, we can make use of the  $\text{cast}^\dagger$  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:

```
let add : Nat  $\rightarrow$  Nat  $\rightarrow$  Nat =  $\mu f : Nat \rightarrow Nat \rightarrow Nat.$ 
   $\lambda n : Nat. \lambda m : Nat.$ 
    (cast↓ n) Nat m ( $\lambda n' : Nat. S\ (f\ n'\ m)$ )
```

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  $\mu$  primitive to write both recursive types and recursion with ease.

## 3. Applications

In this section, we show some large examples using  $\lambda C_\beta$ .

### 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  $\lambda C_\beta$  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 =
   $\mu ev : PLambda Value \rightarrow Value.$ 
     $\lambda e : PLambda Value. \text{case } e \text{ of}$ 
      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 =
   $\lambda e : Value. \text{case } e \text{ of}$ 
    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  $\lambda C_\beta$  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**  $B (a : \star) = \text{One } a \mid \text{Two } (B (\text{PairT } a \ a));$

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

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

**Figure 3.** Construct a perfect binary tree from a list

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

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

## 4. Explicit Calculus of Constructions with Recursion

In this section, we present our core language, the explicit Calculus of Constructions with recursion ( $\lambda C_\beta$ ). Based on the Calculus of Constructions ( $\lambda C$ ),  $\lambda C_\beta$  enjoys the concise syntax with a uniform representation of terms, types and kinds, as well as the expressiveness of dependent types. In order to support general recursion, we bring the fixpoint and recursive type to term and type level respectively, expressed in the same polymorphic  $\mu$ -notation. Since general recursion on the type level breaks the strong normalizing property, the type checker can be stuck by the original implicit conversion rule in  $\lambda C$  when evaluating recursive types. In  $\lambda C_\beta$ , type conversion is no longer inferred automatically but explicitly driven by two new cast primitives. With such explicit type casting semantics, type-level computation becomes deterministic and type checking of  $\lambda C_\beta$  can be decidable without requiring strong normalization.

### 4.1 Syntax

The basic syntax of  $\lambda C_\beta$  is shown in Figure 4, including abstract syntax of expressions, contexts and values. Inherited from  $\lambda C$ ,  $\lambda C_\beta$  uses a single syntactic level to represent terms, types and kinds, while other typed intermediate languages based on System  $F$  or  $F_\omega$  usually distinguish them, e.g. System  $F_c$  of GHC, or  $\mu F^*$  of  $F^*$ . This brings the economy that a single set of rules can be used for terms, types and kinds uniformly so that significantly simplifies the implementation of type checker. We use metavariables  $e$  and  $\tau$  when referring to a “term” and a “type” respectively. Note that without distinction of syntactic levels, we still informally use words term, type and kind. For a certain typing judgement  $\Gamma \vdash e : \tau$ , we call the left-hand-side  $e$  as a term-level expression, and the right-hand-side  $\tau$  as a type-level expression. For example, in  $\sigma : \star$ , the term-level expression  $\sigma$  is traditionally a type with kind  $\star$ .

Similar to  $\lambda C$ ,  $\lambda C_\beta$  uses a product form  $\Pi x : \tau_1. \tau_2$  to represent both traditional and dependent function types. We interchangeably use the arrow form  $(x : \tau_1) \rightarrow \tau_2$  of the product in the source language for brevity. By convention, we also use the syntactic sugar  $\tau_1 \rightarrow \tau_2$  to represent the product if  $x$  does not occur free in  $\tau_2$ .

The syntax difference of from  $\lambda C$  is that  $\lambda C_{\text{exp}}$  introduces two new explicit type conversion primitives, namely  $\text{cast}^\uparrow$  and  $\text{cast}_\downarrow$  (pronounced as “cast up” and “cast down”), in order to replace the implicit conversion rule of  $\lambda C$ . They represent two directions of type conversion operations:  $\text{cast}_\downarrow$  stands for the reduction of types while  $\text{cast}^\uparrow$  is the inverse. Specifically speaking, suppose we have  $e : \sigma$ , i.e. the type of expression  $e$  is  $\sigma$ .  $\text{cast}^\uparrow [\tau] e$  converts the type of  $e$  to  $\tau$ , if there exists a type  $\tau$  such that it can be reduced to  $\sigma$  in a single step, i.e.  $\tau \rightarrow \sigma$ .  $\text{cast}_\downarrow e$  represents the one-step-reduced type of  $e$ , i.e.  $(\text{cast}_\downarrow e) : \sigma'$  if  $\sigma \rightarrow \sigma'$ .

The intention of introducing two explicit cast primitives is that we can gain full control of computation at the type level by manually managing the type conversions. Later in §4.2 we will see dropping the implicit conversion rule of  $\lambda C$  simplifies the type checking and leads to syntax-directed typing rules. This also influences the requirements of decidable type checking, that strong normalization is no long necessary.

### 4.2 Type system

The type system for  $\lambda C_{\text{exp}}$  contains typing judgements and operational semantics. Figure 5 lists operational semantics for  $\lambda C_{\text{exp}}$  that defines rules for one-step reduction, including the  $\beta$ -reduction rule and  $\text{cast}_\downarrow$  rules. The expressions will be reduced by applying rules one or more times. Rule  $S\_CASTDOWN$  prevents the reduction from stalling with  $\text{cast}_\downarrow$  and continues to reduce the inner expression. Rule  $S\_CASTDOWNUP$  states that  $\text{cast}_\downarrow$  cancels the  $\text{cast}^\uparrow$  of an expression.

$e, \tau, \sigma$	$::=$	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 to type
$\text{cast}_\downarrow e$		Cast down by reduction
$\mu x : \tau. e$		General recursion
$\Gamma$	$::=$	Contexts
$\emptyset$		Empty
$\Gamma, x : \tau$		Variable binding
$v$	$::=$	Values
$\lambda x : \tau. e$		Abstraction
$\Pi x : \tau_1. \tau_2$		Product
$\text{cast}^\uparrow [\tau] e$		Cast up

Figure 4. Syntax of  $\lambda C_\beta$

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

Figure 5. Operational semantics of  $\lambda C_{\text{exp}}$

Figure 6 lists the typing judgements to check the validity of expressions. Most rules are straightforward and similar with the ones in  $\lambda C$ . For example, rule T\_AX states that the “type” of sort  $\star$  is a kind. This is derived from an axiom in  $\lambda C$ , that the highest sort is  $\square$ , making the type system predicative. Rule T\_PI allows us to type dependent products. There are four possible combinations of types of  $\tau_1$  and  $\tau_2$  in a product  $\Pi x : \tau_1. \tau_2$ , i.e.  $(s, t) \in \{\star, \square\} \times \{\star, \square\}$ . For some  $(\lambda x : \tau_1. e) : (\Pi x : \tau_1. \tau_2)$ , when  $(s, t) = (\star, \square)$ ,  $x : \tau_1 : \star$ ,  $e : \tau_2 : \square$ , so  $x$  is a term and  $e$  is a type. Thus, we have a type depending on a term which means the product is a dependent type.

The difference from  $\lambda C$  for typing rules of  $\lambda C_{\text{exp}}$  is that rule T\_CASTUP and T\_CASTDOWN are added to check the type conversion primitives  $\text{cast}^\uparrow$  and  $\text{cast}_\downarrow$ , and the implicit type conversion rule of  $\lambda C$  is removed, which is the rule as follows:

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

This rule is necessary for  $\lambda C$  because of the premise requirements of the application rule T\_APP:

$$\frac{\Gamma \vdash e_1 : (\Pi x : \tau_2. \tau_1) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_1[x \mapsto e_2]} \text{ T\_APP}$$

Consider the following two cases of the term  $e_1 e_2$ :

- $e_2$  can be an arbitrary term so its type  $\tau_2$  is not necessary in normal form which might break the type checking of  $e_1$ , e.g. suppose  $e_1 : \sigma \rightarrow \tau$  and  $e_2 : \tau_2$ , where  $\tau_2$  is an application  $(\lambda x : \star. x) \sigma$ . By TCC\_CONV,  $(\lambda x : \star. x) \sigma$  is  $\beta$ -equivalent to  $\sigma$ , thus  $e_2 : \sigma$  and we can further use T\_APP to achieve  $e_1 e_2 : \tau$ .
- The type of  $e_1$  should be a product expression according to the premise. But without the conversion rule, the term fails to type check if the type of  $e_1$  is an expression which can further evaluate to a product, e.g.  $\Pi y : ((\lambda x : \star. x) \tau_2). \tau_1$ . After applying TCC\_CONV, the type of  $e_1$  is converted to its  $\beta$ -equivalence  $\Pi x : \tau_2. \tau_1$ . Thus we can further apply the T\_APP.

We need to show that explicit type conversion rules with cast primitives can also satisfy the premises of rule T\_APP. Still consider the above two cases:

- Given  $e_1 : \sigma \rightarrow \tau$  and  $e_2 : (\lambda x : \star. x) \sigma$ , we do the application by term  $e_1 (\text{cast}_\downarrow e_2)$ . Since  $(\lambda x : \star. x) \sigma \rightarrow \sigma$ ,  $\text{cast}_\downarrow e_2 : \sigma$ , the term  $e_1 (\text{cast}_\downarrow e_2)$  type-checks with the rule T\_APP.
- Given  $e_1 : (\Pi y : ((\lambda x : \star. x) \tau_2). \tau_1)$  and  $e_2 : \tau_2$ , we do the application by term  $e_1 (\text{cast}^\uparrow [(\lambda x : \star. x) \tau_2] e_2)$ . Noting that  $(\lambda x : \star. x) \tau_2 \rightarrow \tau_2$ , the term conforms to rule T\_CASTUP. Thus  $\text{cast}^\uparrow [(\lambda x : \star. x) \tau_2] e_2 : ((\lambda x : \star. x) \tau_2)$  and the term  $e_1 (\text{cast}^\uparrow [(\lambda x : \star. x) \tau_2] e_2)$  can be type-checked by the rule T\_APP.

Therefore, it is feasible to replace implicit conversion rules of  $\lambda C$  with explicit type conversion rules.

$\Gamma \vdash e : \tau$  Typing rules of  $\lambda C$

$\frac{}{\emptyset \vdash \star : \square}$	TCC_AX
$\frac{\Gamma \vdash \tau : s \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau}$	TCC_VAR
$\frac{\Gamma \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 : s \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : \tau_1 \vdash e : \tau_2}$	TCC_WEAK
$\frac{\Gamma \vdash e_1 : (\Pi x : \tau_2. \tau_1) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_1[x \mapsto e_2]}$	TCC_APP
$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash (\Pi x : \tau_1. \tau_2) : s}{\Gamma \vdash (\lambda x : \tau_1. e) : (\Pi x : \tau_1. \tau_2)}$	TCC_LAM
$\frac{\Gamma \vdash \tau_1 : s \quad \Gamma, x : \tau_1 \vdash \tau_2 : t}{\Gamma \vdash (\Pi x : \tau_1. \tau_2) : t}$	TCC_PI
$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_2 : s \quad \tau_1 =_\beta \tau_2}{\Gamma \vdash e : \tau_2}$	TCC_CONV

Figure 6. Typing rules of  $\lambda C$

### 4.3 Decidability and soundness without strong normalization

The conversion rule of  $\lambda C$  is not syntax-directed because it can be implicitly applied at any time in a derivation. The  $\beta$ -equality premise of the rule also leads to the decidability of type checking relying on the strong normalization property of  $\lambda C$ . Suppose strong normalization does not hold in the type system, then we can find a type  $\tau_1$  such that there exists at least one reduction sequence which does not terminate. Notice that any type  $\tau_2$  in such reduction

sequence holds for  $\tau_1 =_\beta \tau_2$ . Thus we can constantly apply the conversion rule without termination and the type checking will not stop, which means the type checking is undecidable.

Requiring strong normalization to achieve the decidability of type checking makes it impossible to combine general recursion with  $\lambda C$ , because general recursion might cause nontermination which simply breaks the strong normalization property. So we use explicit type conversion rules by cast operations to relax the constraints of achieving decidable type checking. We have the following theorem:

**Theorem 4.1** (Decidability of type checking for  $\lambda C_{\text{exp}}$ ). *Let  $\Gamma$  be an environment,  $e$  and  $\tau$  be expressions of  $\lambda C_{\text{exp}}$  such that  $\Gamma \vdash \tau : \star$ . Then the problem of knowing if one has  $\Gamma \vdash e : \tau$  is decidable.*

*Proof.* By induction on typing rules in Figure 6.  $\square$

Notice that new explicit type conversion rules are syntax-directed and do not include the  $\beta$ -equality premise but one-step reduction instead. Because checking if one term is one-step-reducible to the other is always decidable by enumerating the reduction rules, type checking using these rules are always decidable. Therefore the proof of decidability for  $\lambda C_{\text{exp}}$  does not rely on the strong normalization. This also implies the possibility of introducing general recursion into the system with decidable type checking.

Also for obtaining the soundness of  $\lambda C_{\text{exp}}$ , the proof does not need the strong normalization by combining the following two theorems:

**Theorem 4.2** (Subject Reduction). *If  $\Gamma \vdash e : \tau$  and  $e \rightarrow e'$  then  $\Gamma \vdash e' : \tau$ .*

*Proof.* By induction on rules in Figure 5.  $\square$

**Theorem 4.3** (Progress). *If  $\emptyset \vdash e : \tau$  then either  $e$  is a value  $v$  or there exists  $e'$  such that  $e \rightarrow e'$ .*

*Proof.* By induction on rules in Figure 6.  $\square$

## 5. The Explicit Calculus of Constructions with Recursion

**BRUNO:** Linus and Jeremy, I think you should do this section together. Most work is on Linus though since he needs to work out the proofs. Jeremy is mostly for Linus to consult with here :).

We have shown that  $\lambda C_{\text{exp}}$  does not rely on strong normalization for decidable type checking and soundness. Thus it is safe to combine general recursion with  $\lambda C_{\text{exp}}$  under the control of explicit type conversion operations  $\text{cast}^\uparrow$  and  $\text{cast}_\downarrow$ . We extend  $\lambda C_{\text{exp}}$  into  $\lambda C_\beta$  by introducing one unified primitive called  $\mu$ -notation for general recursion. It functions as a fixed point at the term level as well as a recursive type at the type level.

### 5.1 The $\mu$ -notation

Based on the syntax of  $\lambda C_{\text{exp}}$ , we add the following  $\mu$ -notation for  $\lambda C_\beta$  (the same part as  $\lambda C_{\text{exp}}$  is left out):

$e, \tau$	$::=$	Expressions
$\mu x : \tau. e$	$\dots$	General recursion

The  $\mu$ -notation is similar to the definition of recursive types, except that it is not only treated as types but also terms. This also corresponds to the property of  $\lambda C_{\text{exp}}$  that terms and types are not distinguished.

The typing rule and operational semantics of  $\mu$ -notation for terms and types are also unified, thus each one rule for static and dynamic semantics is only needed to add over  $\lambda C_{\text{exp}}$ . The new type checking rule of  $\mu$ -notation is as follows:

$$\frac{\Gamma, x : \tau \vdash e : \tau \quad \Gamma \vdash \tau : \star}{\Gamma \vdash (\mu x : \tau. e) : \tau} \text{ T\_MU}$$

And the one-step reduction rule is as follows:

$$\frac{}{\mu x : \tau. e \rightarrow e[x \mapsto \mu x : \tau. e]} \text{ S\_MU}$$

If  $\mu x : \tau. e$  is a term, with the S\_MU rule, it is not treated as a value and can be further reduced, which is different from conventional iso-recursive types. The one-step reduced term of  $\mu x : \tau. e$  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  $\beta$ -equivalence. But such result cannot be directly obtained because of the removal of implicit conversion rule. Instead, by using explicit cast operations of  $\lambda C_{\text{exp}}$ , we can obtain the following transformation between  $e$  and  $e'$ :

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

For type-level  $\mu$ -notation,  $\text{cast}^\uparrow$  and  $\text{cast}_\downarrow$  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,  $\lambda C_\beta$  is no long consistent so that not able to be used as a logic. But with the power of general recursion, the expressibility of  $\lambda C_\beta$  is increased since more data types and functions can be mapped or encoded into  $\lambda C_\beta$ . And more importantly, even with  $\mu$ -notation,  $\lambda C_\beta$  can still be proved to have the same properties as  $\lambda C_\beta$  in the sense of decidability of type checking and soundness.

As what we previously illustrate in Section 4.3, the type checking of  $\lambda C_{\text{exp}}$  can always terminate because the derivation is finite without the implicit conversion rule. With the  $\mu$ -notation in  $\lambda C_\beta$ , 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  $\text{cast}^\uparrow$  and  $\text{cast}_\downarrow$ , 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 ( $\lambda C_{\text{suf}}$ ) that supports simple datatypes and case analysis. Due to the expressiveness of  $\lambda C_{\beta}$ , all these features can be elaborated into the core language without extending the built-in language constructs of  $\lambda C_{\beta}$ . In what follows, we first give the syntax of  $\lambda C_{\text{suf}}$ , followed by the extended typing rules, then we show the formal translation rules that translates  $\lambda C_{\text{suf}}$  expressions into  $\lambda C_{\beta}$  expressions. Finally we demonstrate the translation using a simple example.

### 6.1 Extended Syntax

The syntax of  $\lambda C_{\text{suf}}$  is shown in Figure 7. Compared with  $\lambda C_{\beta}$ ,  $\lambda C_{\text{suf}}$  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  $\Pi$  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		
$u$	$::= x \mid K$	Variables and constructors
$e, \tau, \sigma, \nu, \kappa$	$::= u$	Term atoms
	$\dots$	
$p$	$\mid \mathbf{case} \ e \ \mathbf{of} \ \overline{p \Rightarrow e}$	Case analysis
	$::= K \ \bar{x} : \bar{\tau}$	Pattern
Environments		
$\Gamma$	$::= \emptyset$	Empty
	$\mid \Gamma, u : \tau$	Variable binding

**Figure 7.** Syntax of  $\lambda C_{\text{suf}}$  ( $e$  for terms;  $\tau, \sigma, \nu$  for types;  $\kappa$  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  $\lambda C_{\text{suf}}$  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 universally quantified type variables with proper concrete types.

### 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  $\lambda C_{\beta}$  expression  $E$  is the translation of  $\lambda C_{\text{suf}}$  expression  $e$  of type  $\tau$ . 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  $\lambda C_{\beta}$  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 an 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  $\lambda C_{\beta}$  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 datatype.

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 gadt: system f with first-class equality proofs. *Higher-Order and Symbolic Computation*, 23(2):145–166, 2010.

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

Figure 9. Typing rules of  $\lambda C_{\text{suf}}$

- [8] J.-W. Roorda 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. Cray, 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. Specification of core language

### A.1 Syntax

$e, \tau, \sigma$	$::=$	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 to type
$\text{cast}_\downarrow e$		Cast down by reduction
$\mu x : \tau. e$		General recursion
$\Gamma$	$::=$	Contexts
$\emptyset$		Empty
$\Gamma, x : \tau$		Variable binding

$v$	$::=$	Values
$\lambda x : \tau. e$		Abstraction
$\Pi x : \tau_1. \tau_2$		Product
$\text{cast}^\uparrow [\tau] e$		Cast up

### A.2 Operational semantics and expression typing

$\boxed{e \longrightarrow e'}$	One-step reduction
$\overline{(\lambda x : \tau. e_1) e_2 \longrightarrow e_1[x \mapsto e_2]}$	S_BETA
$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$	S_APP
$\frac{e \longrightarrow e'}{\text{cast}_\downarrow e \longrightarrow \text{cast}_\downarrow e'}$	S_CASTDOWN
$\overline{\text{cast}_\downarrow (\text{cast}^\uparrow [\tau] e) \longrightarrow e}$	S_CASTDOWNUP
$\overline{\mu x : \tau. e \longrightarrow e[x \mapsto \mu x : \tau. e]}$	S_MU
$\boxed{\vdash \Gamma}$	Well-formed context
$\overline{\vdash \emptyset}$	ENV_EMPTY
$\frac{\vdash \Gamma \quad \Gamma \vdash \tau : \star}{\vdash \Gamma, x : \tau}$	ENV_VAR
$\boxed{\Gamma \vdash e : \tau}$	Expression typing
$\overline{\emptyset \vdash \star : \star}$	T_AX
$\frac{\vdash \Gamma \quad x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$	T_VAR
$\frac{\Gamma \vdash e_1 : (\Pi x : \tau_2. \tau_1) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_1[x \mapsto e_2]}$	T_APP
$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash (\Pi x : \tau_1. \tau_2) : \star}{\Gamma \vdash (\lambda x : \tau_1. e) : (\Pi x : \tau_1. \tau_2)}$	T_LAM
$\frac{\Gamma \vdash \tau_1 : \star \quad \Gamma, x : \tau_1 \vdash \tau_2 : \star}{\Gamma \vdash (\Pi x : \tau_1. \tau_2) : \star}$	T_PI
$\frac{\Gamma \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 : \star \quad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\text{cast}^\uparrow [\tau_1] e) : \tau_1}$	T_CASTUP



$$\boxed{\Gamma \vdash e : \tau \rightsquigarrow E}$$

(Ax)

$$\overline{\emptyset \vdash \star : \square \rightsquigarrow \star}$$

(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)

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow E \quad \Gamma \vdash (\Pi x : \tau_1. \tau_2) : t}{\Gamma \vdash (\lambda x : \tau_1. e) : (\Pi x : \tau_1. \tau_2) \rightsquigarrow \lambda x : \tau_1. E} \quad t \in \{\star, \square\}$$

(Pi)

$$\frac{\Gamma \vdash \tau_1 : s \quad \Gamma, x : \tau_1 \vdash \tau_2 : t}{\Gamma \vdash (\Pi x : \tau_1. \tau_2) : t \rightsquigarrow \Pi x : \tau_1. \tau_2} \quad (s, t) \in \mathcal{R}$$

(Mu)

$$\frac{\Gamma, x : \tau \vdash e : \tau \rightsquigarrow E \quad \Gamma \vdash \tau : s}{\Gamma \vdash (\mu x : \tau. e) : \tau \rightsquigarrow \mu x : \tau. E} \quad s \in \{\star, \square\}$$

(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)

$$\frac{\Gamma \vdash e_1 : \sigma \rightsquigarrow E_1 \quad \Gamma \vdash_p p \Rightarrow e_2 : \sigma \rightarrow \tau \rightsquigarrow E_2}{\Gamma \vdash \text{case } e_1 \text{ of } \bar{p} \Rightarrow \bar{e}_2 : \tau \rightsquigarrow (\text{cast}_\downarrow E_1) \tau \bar{E}_2}$$

$$\boxed{\Gamma \vdash_p p \Rightarrow e : \sigma \rightarrow \tau \rightsquigarrow E}$$

(Alt)

$$\frac{\theta = [\bar{u} := \bar{v}]}{K : \Pi \bar{u} : \bar{\kappa}. \bar{\sigma} \rightarrow D \bar{u} \in \Gamma \quad \Gamma, x : \theta(\sigma) \vdash e : \tau \rightsquigarrow E} \quad \Gamma \vdash_p K x : \theta(\sigma) \Rightarrow e : D \bar{v} \rightarrow \tau \rightsquigarrow \lambda(x : \theta(\sigma)). E$$

$$\boxed{\Gamma \vdash \text{decl} : \Gamma_d \rightsquigarrow E}$$

(Data)

$$\frac{\Gamma \vdash \bar{\kappa} \rightarrow \star : \square \quad \Gamma, D : \bar{\kappa} \rightarrow \star, \bar{u} : \bar{\kappa} \vdash \bar{\tau} \rightarrow D \bar{u} : \star}{\Gamma \vdash (\text{data } D \bar{u} : \bar{\kappa} = \bar{K} \bar{\tau}) : (D : \bar{\kappa} \rightarrow \star, \bar{K} : \Pi \bar{u} : \bar{\kappa}. \bar{\tau} \rightarrow D \bar{u}) \rightsquigarrow E} \\ E ::= \text{let } D : \bar{\kappa} \rightarrow \star = \lambda \bar{u} : \bar{\kappa}. \mu X : \star. \Pi b : \star. (\bar{\tau}[D \bar{u} := X] \rightarrow b) \rightarrow b \text{ in} \\ \text{let } K_i : \Pi \bar{u} : \bar{\kappa}. \bar{\tau} \rightarrow D \bar{u} = \lambda(u : \bar{\kappa}). \lambda(x : \bar{\tau}). \\ \text{cast}^\uparrow[D \bar{u}] (\lambda(b : \star)(c : \bar{\tau} \rightarrow b). c_i \bar{x}) \text{ in}$$

$$\boxed{\Gamma \vdash \text{pgm} : \tau \rightsquigarrow E}$$

(Pgm)

$$\frac{\Gamma_0 \vdash \text{decl} : \Gamma_d \rightsquigarrow E_1 \quad \Gamma = \Gamma_0, \bar{\Gamma}_d \quad \Gamma \vdash e : \tau \rightsquigarrow E}{\Gamma_0 \vdash \text{decl}; e : \tau \rightsquigarrow E_1 \oplus E}$$

**Figure 10.** Type-directed translation from  $\lambda C_{\text{suf}}$  to  $\lambda C_\beta$

## B. Specification of source language

### B.1 Syntax

See Figure 11.

### B.2 Expression typing

See Figure 12.

### B.3 Translation to the core

See Figure 13.

## C. Proofs about core language

### C.1 Properties

**Lemma C.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  $\lambda C$  [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.  $\square$

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

Figure 11. Syntax of source language

**Lemma C.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.

**Case T\_CASTUP:**  $\frac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau_1 \quad \tau_1 \rightarrow \tau_2}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\text{cast}^\uparrow [\tau_1] e_1) : \tau_1}$

By induction hypothesis, we have  $\Gamma_1, \Gamma_2^* \vdash e_1^* : \tau_1^*$ ,  $\Gamma_1, \Gamma_2^* \vdash \tau_1^* : \star$  and  $\tau_1 \rightarrow \tau_2$ . By the definition of substitution, we can obtain  $\tau_1^* \rightarrow \tau_2^*$  by  $\tau_1 \rightarrow \tau_2$ . 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:**  $\frac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau_1 \quad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau_2 : \star \quad \tau_1 \rightarrow \tau_2}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\text{cast}_\downarrow e_1) : \tau_2}$

By induction hypothesis, we have  $\Gamma_1, \Gamma_2^* \vdash e_1^* : \tau_1^*$ ,  $\Gamma_1, \Gamma_2^* \vdash \tau_2^* : \star$  and  $\tau_1 \rightarrow \tau_2$  thus  $\tau_1^* \rightarrow \tau_2^*$ . 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.  $\square$

**Lemma C.3** (Generation lemma).

- (1) If  $\Gamma \vdash x : \sigma$ , then there exist an expression  $\tau$  and a sort  $\star$  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  $\tau_1$  and  $\tau_2$  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 a sort  $\star$  and an expression  $\tau_2$  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  $\Gamma, x : \tau_1 \vdash \tau_2 : \star$ .

$\boxed{\vdash \Gamma}$  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

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

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

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

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

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

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

$$\frac{}{\emptyset \vdash \star : \star} \text{TS\_AX}$$

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

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

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

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

$$\frac{\Gamma \vdash E : T_2 \quad \Gamma \vdash T_1 : \star \quad T_1 \longrightarrow T_2}{\Gamma \vdash (\text{cast}^\uparrow [T_1] E) : T_1} \text{TS\_CASTUP}$$

$$\frac{\Gamma \vdash E : T_1 \quad \Gamma \vdash T_2 : \star \quad T_1 \longrightarrow T_2}{\Gamma \vdash (\text{cast}_\downarrow E) : T_2} \text{TS\_CASTDOWN}$$

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

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

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

**Figure 12.** Typing rules of source language

- (5) If  $\Gamma \vdash (\mu x : \tau. e) : \sigma$ , then there exists a sort  $\star$  such that  $\Gamma \vdash \tau : \star$ ,  $\sigma \equiv \tau$  and  $\Gamma, x : \tau \vdash e : \tau$ .
- (6) If  $\Gamma \vdash (\text{cast}^\uparrow [\tau_1] e) : \sigma$ , then there exist an expression  $\tau_2$  and a sort  $\star$  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  $\tau_1, \tau_2$  and a sort  $\star$  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;

$$\boxed{\Gamma \vdash \text{pgm} : T \rightsquigarrow e}$$

Program translation

$$\frac{\overline{\Gamma_0 \vdash \text{decl} : \Gamma' \rightsquigarrow e_1} \quad \Gamma = \Gamma_0, \overline{\Gamma'} \quad \Gamma \vdash E : T \rightsquigarrow e}{\Gamma_0 \vdash (\overline{\text{decl}}; E) : T \rightsquigarrow \overline{e_1} \uplus e} \text{TRPGM\_PGM}$$

$$\boxed{\Gamma \vdash \text{decl} : \Gamma' \rightsquigarrow e}$$

Datatype translation

$$\frac{\Gamma \vdash \overline{\kappa} \rightarrow \star : \star \rightsquigarrow \overline{\sigma} \rightarrow \star \quad \overline{\Gamma, D : \overline{\kappa} \rightarrow \star, \overline{u} : \overline{\kappa} \vdash \overline{T} \rightarrow D\overline{u} : \star \rightsquigarrow \overline{\tau} \rightarrow D\overline{u}}}{\Gamma \vdash (\text{data } D\overline{u} : \overline{\kappa} = K\overline{T}) : (D : \overline{\kappa} \rightarrow \star, K : (\overline{u} : \overline{\kappa}) \rightarrow \overline{T} \rightarrow D\overline{u}) \rightsquigarrow e} \text{TRDECL\_DATA}$$

$$e \triangleq \text{let } D : \overline{\sigma} \rightarrow \star = \mu X : \overline{\sigma} \rightarrow \star. \lambda \overline{u} : \overline{\sigma}. (\alpha : \star) \rightarrow \overline{(\tau[D \mapsto X] \rightarrow \alpha)} \rightarrow \alpha \text{ in}$$

$$\text{let } K_i : (\overline{u} : \overline{\sigma}) \rightarrow \overline{\tau} \rightarrow D\overline{u} = \lambda \overline{u} : \overline{\sigma}. \lambda \overline{x} : \overline{\tau}. \text{cast}^\uparrow [D\overline{u}] (\lambda \alpha : \star. \lambda b : \overline{\tau} \rightarrow \alpha. b_i \overline{x}) \text{ in}$$

$$\boxed{\Gamma \vdash p \Rightarrow E : S \rightarrow T \rightsquigarrow e}$$

Pattern translation

$$\frac{K : (\overline{u} : \overline{\kappa}) \rightarrow \overline{S} \rightarrow D\overline{u} \in \Gamma \quad \Gamma, x : S[\overline{u} \mapsto \overline{v}] \vdash E : T \rightsquigarrow e \quad \Gamma \vdash S[\overline{u} \mapsto \overline{v}] : \star \rightsquigarrow \sigma}{\Gamma \vdash Kx : S[\overline{u} \mapsto \overline{v}] \Rightarrow E : D\overline{v} \rightarrow T \rightsquigarrow \lambda \overline{x} : \overline{\sigma}. e} \text{TRPAT\_ALT}$$

$$\boxed{\Gamma \vdash E : T \rightsquigarrow e}$$

Expression translation

$$\frac{}{\emptyset \vdash \star : \star \rightsquigarrow \star} \text{TR\_AX}$$

$$\frac{\vdash \Gamma \quad x : T \in \Gamma}{\Gamma \vdash x : T \rightsquigarrow x} \text{TR\_VAR}$$

$$\frac{\Gamma \vdash E_1 : (\Pi x : T_2. T_1) \rightsquigarrow e_1 \quad \Gamma \vdash E_2 : T_2 \rightsquigarrow e_2}{\Gamma \vdash E_1 E_2 : T_1[x \mapsto E_2] \rightsquigarrow e_1 e_2} \text{TR\_APP}$$

$$\frac{\Gamma, x : T_1 \vdash E : T_2 \rightsquigarrow e \quad \Gamma \vdash (\Pi x : T_1. T_2) : \star \rightsquigarrow \Pi x : \tau_1. \tau_2}{\Gamma \vdash (\lambda x : T_1. E) : (\Pi x : T_1. T_2) \rightsquigarrow \lambda x : \tau_1. e} \text{TR\_LAM}$$

$$\frac{\Gamma \vdash T_1 : \star_1 \rightsquigarrow \tau_1 \quad \Gamma, x : T_1 \vdash T_2 : \star_2 \rightsquigarrow \tau_2}{\Gamma \vdash (\Pi x : T_1. T_2) : \star_2 \rightsquigarrow \Pi x : \tau_1. \tau_2} \text{TR\_PI}$$

$$\frac{\Gamma \vdash E : T_2 \rightsquigarrow e \quad \Gamma \vdash T_1 : \star \rightsquigarrow \tau_1 \quad T_1 \longrightarrow T_2}{\Gamma \vdash (\text{cast}^\uparrow [T_1] E) : T_1 \rightsquigarrow \text{cast}^\uparrow [\tau_1] e} \text{TR\_CASTUP}$$

$$\frac{\Gamma \vdash E : T_1 \rightsquigarrow e \quad \Gamma \vdash T_2 : \star \rightsquigarrow \tau_2 \quad T_1 \longrightarrow T_2}{\Gamma \vdash (\text{cast}_\downarrow E) : T_2 \rightsquigarrow \text{cast}_\downarrow e} \text{TR\_CASTDOWN}$$

$$\frac{\Gamma, x : T \vdash E : T \rightsquigarrow e \quad \Gamma \vdash T : \star \rightsquigarrow \tau}{\Gamma \vdash (\mu x : T. E) : T \rightsquigarrow \mu x : \tau. e} \text{TR\_MU}$$

$$\frac{\Gamma \vdash E_1 : S \rightsquigarrow e_1 \quad \overline{\Gamma \vdash p \Rightarrow E_2 : S \rightarrow T \rightsquigarrow e_2} \quad \overline{\Gamma \vdash S \rightarrow T : \star \rightsquigarrow \sigma \rightarrow \tau}}{\Gamma \vdash \text{case } E_1 \text{ of } \overline{p \Rightarrow E_2} : T \rightsquigarrow (\text{cast}_\downarrow e_1) \tau \overline{e_2}} \text{TR\_CASE}$$

$$\frac{\Gamma \vdash E_2 : T \rightsquigarrow e_2 \quad \Gamma \vdash E_1[x \mapsto E_2] : S \rightsquigarrow e_1[x \mapsto e_2]}{\Gamma \vdash \text{let } x : T = E_2 \text{ in } E_1 : S \rightsquigarrow e_1[x \mapsto e_2]} \text{TR\_LET}$$

**Figure 13.** Translation rules of source language

- 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.

can be shown that the statement of the lemma holds and is the only possible case.  $\square$

**Lemma C.4** (Correctness of types). *If  $\Gamma \vdash e : \tau$  then there exists a sort  $\star$  such that  $\tau \equiv \star$  or  $\Gamma \vdash \tau : \star$ .*

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

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

## C.2 Decidability of type checking

**Lemma C.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  $e = \star$ , or  $e = x$ :** No such  $e'$  exists since it is impossible to reduce a sort or a variable.

**Case  $e = v$ :**  $e$  has one of the following forms: (1)  $\lambda x : \tau. e$ , (2)  $\Pi x : \tau_1. \tau_2$ , (3)  $\text{cast}^\uparrow[\tau]e$ , 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.

**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  $\lambda$ -term:** If  $e_1 = v$ , there is no  $e'_1$  such that  $e_1 \longrightarrow e'_1$ . Since  $e_1$  is not a  $\lambda$ -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  $\text{cast}^\uparrow$ -term:** If  $e_1 = v$ , there is no  $e'_1$  such that  $e_1 \longrightarrow e'_1$ . Since  $e_1$  is not a  $\text{cast}^\uparrow$ -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 C.6** (Decidability of type checking). *There is a decidable algorithm which given  $\Gamma, e$  computes the unique  $\tau$  such that  $\Gamma \vdash e : \tau$  or reports there is no such  $\tau$ .*

*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  $\Gamma$  that is well-formed. By rule TS.VAR, if  $x : \tau \in \Gamma$ ,  $\tau$  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 C.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  $\tau_2$ , and check whether  $\tau_1$  is legal, i.e. its sorts is  $\star$ . If  $\tau_1$  is legal, by Lemma C.5, there is at most one  $\tau'_1$  such that  $\tau_1 \longrightarrow \tau'_1$ . If such  $\tau'_1$  does not exist, then we report the type checking is failed. Otherwise, we examine if  $\tau'_1$  is syntactically equal to  $\tau_2$ , i.e.  $\tau'_1 \equiv \tau_2$ . If the equality holds, we obtain the unique type of  $e$  which is  $\tau_1$ . 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  $\tau_1$ . By Lemma C.5, there is at most one  $\tau_2$  such that  $\tau_1 \longrightarrow \tau_2$ . If such  $\tau_2$  exists and its sorts is  $\star$ , we have found the unique type of  $e$  is  $\tau_2$ . Otherwise, we report  $e$  fails to type check.

□

## C.3 Soundness

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

**Lemma C.8** (Subject reduction). *If  $\Gamma \vdash e : \sigma$  and  $e \twoheadrightarrow e'$  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  $e \twoheadrightarrow e'$ . The proof is by induction with respect to the definition of one-step reduction  $\longrightarrow$  as follows:

**Case  $\frac{}{(\lambda x : \tau. e_1) e_2 \longrightarrow e_1[x \mapsto e_2]}$  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 C.3(2), there exist expressions  $\tau'_1$  and  $\tau_2$  such that

$$\Gamma \vdash (\lambda x : \tau_1. e_1) : (\Pi x : \tau'_1. \tau_2) \quad (1)$$

$$\Gamma \vdash e_2 : \tau'_1$$

$$\sigma \equiv \tau_2[x \mapsto e_2]$$

By Lemma C.3(3), the judgement (1) implies that there exists an expression  $\tau'_2$  such that

$$\Pi x : \tau'_1. \tau_2 \equiv \Pi x : \tau_1. \tau'_2 \quad (2)$$

$$\Gamma, x : \tau_1 \vdash e_1 : \tau'_2$$

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 C.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  $\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$  S.APP:**

Suppose  $\Gamma \vdash e_1 e_2 : \sigma$  and  $\Gamma \vdash e'_1 e_2 : \sigma'$ . By Lemma C.3(2), there exist expressions  $\tau_1$  and  $\tau_2$  such that

$$\Gamma \vdash e_1 : (\Pi x : \tau_1. \tau_2)$$

$$\Gamma \vdash e_2 : \tau_1$$

$$\sigma \equiv \tau_2[x \mapsto e_2]$$

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  $\frac{e \longrightarrow e'}{\text{cast}_\downarrow e \longrightarrow \text{cast}_\downarrow e'}$  S.CASTDOWN:**

Suppose  $\Gamma \vdash \text{cast}_\downarrow e : \sigma$  and  $\Gamma \vdash \text{cast}_\downarrow e' : \sigma'$ . By Lemma C.3(7), there exist expressions  $\tau_1, \tau_2$  and a sort  $\star$  such that

$$\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_2 : \star$$

$$\tau_1 \longrightarrow \tau_2 \quad \sigma \equiv \tau_2$$

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  $\frac{}{\text{cast}_\downarrow(\text{cast}^\uparrow[\tau]e) \longrightarrow e}$  S.CASTDOWNUP:**

Suppose  $\Gamma \vdash \text{cast}_\downarrow(\text{cast}^\uparrow[\tau]e) : \sigma$  and  $\Gamma \vdash e : \sigma'$ . By Lemma C.3(7), there exist expressions  $\tau'_1, \tau_2$  such that

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

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

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

By Lemma C.3(6), the judgement (3) implies that there exists an expression  $\tau'_2$  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 C.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 C.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 C.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  $\tau$ , 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 C.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**  $e = \star$ : Trivial by rule T\_AX where  $\sigma \equiv \star$ .

**Case**  $e = x$ : Impossible, since the context is empty.

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

**Case**  $e = e_1 e_2$ : By Lemma C.3(2), there exist expressions  $\tau_1$  and  $\tau_2$  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 C.3(3), it must be a  $\lambda$ -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 C.3(7), there exist expressions  $\tau_1$  and  $\tau_2$  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 C.3(6), it must be a  $\text{cast}^\uparrow$ -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  $e' \equiv e_1[x \mapsto \mu x : \tau. e_1]$ .

□