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 langauge

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. Indeed the core language for modern versions of functional languages are quite complex, having multiple syntactic sorts (such as terms, types and kinds), as well as dozens of language constructs []BRUNO: F_C.

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. There is a clear tension between decidability of type-checking and allowing general recursion at the type-level.

This paper proposes λC_β: a variant of the calculus of constructions allows the same syntax for types and terms, supports type-level computation, and preserves decidability of type-checking under the presence of general recursion. In λC_β, each type-level computation step is explicit. BRUNO: emphasis on the advantages: a minimal core language? The key idea, which is inspired by the traditional treatment of *iso-recursive types*, is to introduce each beta reduction or expansion at the type-level by a *type-safe cast*. The casts allow control over the type-level computation. For example, if a type-level program requires two beta-reductions to reach normal form, then two casts are needed in the program. If a non-terminating program is used at the type-level, it is not possible

to cause non-termination in the type-checker, because that would require a program with an infinite number of casts. Since single beta-steps are trivially terminating, decidability of type-checking is possible even in the presence of non-terminating programs at the type-level.

Although it may seem that forcing each step of computation at the type-level to be explicit will prevent convinient use of type-level computation.

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...

The paper also shows how many of programming language features of Haskell, including some of the latest extensions, can be encoded in λC_β via a source language. In particular the source language supports *algebraic datatypes*, *higher-kinded types*, *nested datatypes*, *kind polymorphism* and *datatype promotion*. This result is interesting because λC_β is a minimal calculus with only 8 language constructs and a single syntactic sort. In contrast the latest versions of System F_C (Haskell's core language) have multiple syntactic sorts and dozens of language constructs. Even if support for equality and coercions, which constitutes a significant part of System F_C, would be removed the resulting language would still be significantly larger and more complex than λC_β.

In summary, the constributions of this work are:

- Decidable type checking and managed type-level computation by replacing implicit conversion rule of CoC with generalized fold/unfold semantics.
- A core language based on Calculus of Constructions (CoC) that collapses terms, types and kinds into the same hierarchy, supports general recursion...
- General recursion by introducing recursive types for both terms and types by the same μ primitive.
- Surface language that supports datatypes, pattern matching and other language extensions for Haskell, and can be encoded into the core language.

a) Motivations:

- Because of the reluctance to introduce dependent types[1], the current intermediate language of Haskell, namely System F_C [**?**], separates expressions as terms, types and kinds, which brings complexity to the implementation as well as further extensions [**? ?**].
- 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 [**?**] and one of its implementation [**?**] show the simplicity of the Pure Type System (PTS). [**?**] also tries to combine recursion with PTS.
- Zombie [**? ?**] is a language with two fragments supporting logics with non-termination. It limits the β-reduction for congruence closure [**?**].
- $\Pi\Sigma$ [**?**] is a simple, dependently-typed core language for expressing high-level constructions[2]. UHC compiler [**?**] tries to use a simplified core language with coercion to encode GADTs.
- System F_C [**?**] has been extended with type promotion [**?**] and kind equality [**?**]. The latter one introduces a limited form of dependent types into the system[3], 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 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) [**?**], which is a higher-order typed lambda calculus. One "unconventional" feature of λC is the so-called *conversion* rule as shown below:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_2 : s \qquad \tau_1 =_\beta \tau_2}{\Gamma \vdash e : \tau_2} \quad \textsc{Tcc\_Conv}$$

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

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

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

2.2 Explicit Type Conversion Rules

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

[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 langauge.

[3] Richard A. Eisenberg is going to implement kind equality [**?**] 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`.

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: $\mathsf{cast}^\uparrow$ and $\mathsf{cast}_\downarrow$.

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

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

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

The dual operation of $\mathsf{cast}^\uparrow$ is $\mathsf{cast}_\downarrow$. The use of $\mathsf{cast}_\downarrow$ is better explained by another similar example. Suppose that

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

and term z has type

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

$g\,z$ is again an ill-typed application, while $g\,(\mathsf{cast}_\downarrow\,z)$ is type correct because $\mathsf{cast}_\downarrow$ 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 [**?**]. 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

$$d : \mathsf{Int} \to \star$$

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

$$d\,\mathsf{loop}$$

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 $d\,\mathsf{loop}$, 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 $\mathsf{cast}_\downarrow$'s to make the type checker loop forever (e.g., $(\lambda x : d\,3.x)(\mathsf{cast}_\downarrow(\mathsf{cast}_\downarrow \ldots z))$, 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-recusive* approach treats a recursive type and its unfolding as different, but isomorphic. In λC_β, this is witnessed by first $\mathsf{cast}^\uparrow$, then $\mathsf{cast}_\downarrow$. A classic example of recursive types is the so-called "hungry" type: $H = \mu\sigma : \star.\,\mathsf{Int} \to \sigma$. 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:

$$\mathsf{cast}_\downarrow\,z : \mathsf{Int} \to H$$
$$\mathsf{cast}_\downarrow\,(\mathsf{cast}_\downarrow\,z) : \mathsf{Int} \to \mathsf{Int} \to H$$
$$\mathsf{cast}_\downarrow(\mathsf{cast}_\downarrow \ldots z) : \mathsf{Int} \to \mathsf{Int} \to \cdots \to H$$

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

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

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 $\mu f : T \to T.\,E$ (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 recusive functions using datatypes. While inductive datatypes can be encoded using either the Church or the Scott encoding, we adopt the Scott encoding as it is bear 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:

$$\mathbf{data}\ \mathit{Nat} = Z \mid S\ \mathit{Nat};$$

In the Scoot 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 $\mu\ X : \star.\,\Pi\ B : \star.\,B \to (X \to B) \to B$.

As can be seen, in the function type $B \to (X \to B) \to B$, B corresponds to the type of the Z constructor, and $X \to B$ 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:

$$
\begin{array}{l}
\textbf{let } Z : Nat = \mathsf{cast}^{\uparrow}\ [Nat]\ (\lambda B : \star.\, \lambda z : B.\, \lambda f : Nat \to B.\, z) \\
\textbf{in} \\
\textbf{let } S : Nat \to Nat = \lambda n : Nat. \\
\quad \mathsf{cast}^{\uparrow}\ [Nat]\ (\lambda B : \star.\, \lambda z : B.\, \lambda f : Nat \to B.\, f\ n) \\
\textbf{in}
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{let } add : Nat \to Nat \to Nat = \mu\, f : Nat \to Nat \to Nat. \\
\quad \lambda n : Nat.\, \lambda m : Nat. \\
\qquad (\mathsf{cast}_{\downarrow}\ n)\ Nat\ m\ (\lambda n' : Nat.\, S\ (f\ n'\ m))
\end{array}
$$

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 §5.)

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 applications, which either Haskell needs non-trivial extensions to do that, or languages like Coq and Agda are impossible to do, whereas we can easily achieve in λC_{β}.

Conventional datatypes Conventional datatypes like natural numbers or polymorphic lists can be easily defined in λC_{β}, as in Haskell. For example, below is the definition of polymorphic lists:

$$
\textbf{data } List\ (a : \star) = Nil \mid Cons\ a\ (List\ a);
$$

Because λC_{β} is explicitly typed, each type parameter needs to be accompanied with corresponding kind expressions. The use of the above datatype is best illustrated by the *length* function:

$$
\begin{array}{l}
\textbf{letrec } length : (a : \star) \to List\ a \to nat = \\
\quad \lambda a : \star.\, \lambda l : List\ a.\, \textbf{case } l \textbf{ of} \\
\qquad Nil \Rightarrow 0 \\
\quad \mid Cons\ (x : a)\ (xs : List\ a) \Rightarrow \\
\qquad 1 + length\ a\ xs \\
\textbf{in} \\
\textbf{let } test : List\ nat = Cons\ nat\ 1\ (Cons\ nat\ 2\ (Nil\ nat)) \\
\textbf{in } length\ nat\ test \quad \text{-- return 2}
\end{array}
$$

HOAS *Higher-order abstract syntax* is a generalization of representing programs where the function space of the meta-language is used to encode the binders of the object language. Because the recursive mention of the datatype can appear in a negative position, systems like Coq and Agda would reject programs using HOAS due to the restrictiveness of their termination checkers. However λC_{β} is able to express HOAS in a straightforward way. We show an example of encoding simply typed lambda calculus:

$$
\begin{array}{l}
\textbf{data } Exp = Num\ nat \\
\quad \mid Lam\ (Exp \to Exp) \\
\quad \mid App\ Exp\ Exp;
\end{array}
$$

Next we define the evaluator for our lambda calculus. As noted by [], the evaluation function needs an extra function *reify* to invert the result of evaluation. The code is presented in Figure 1.

$$
\begin{array}{l}
\textbf{data } Value = VI\ nat \mid VF\ (Value \to Value); \\
rec\ Eval = Ev\ \{eval' : Exp \to Value, reify' : Value \to Exp\}; \\
\textbf{let } f : Eval = \mu\, f' : Eval. \\
\quad Ev\ (\lambda e : Exp.\, \textbf{case } e \textbf{ of} \\
\qquad Num\ (n : nat) \Rightarrow VI\ n \\
\qquad \mid Lam\ (fun : Exp \to Exp) \Rightarrow \\
\qquad\quad VF\ (\lambda e' : Value.\, eval'\ f'\ (fun\ (reify'\ f'\ e'))) \\
\qquad \mid App\ (a : Exp)\ (b : Exp) \Rightarrow \\
\qquad\quad \textbf{case } eval'\ f'\ a \textbf{ of} \\
\qquad\qquad VI\ (n : nat) \Rightarrow VI\ n \quad \text{-- abnormal branch} \\
\qquad\qquad \mid VF\ (fun : Value \to Value) \Rightarrow fun\ (eval'\ f'\ b)) \\
\qquad (\lambda v : Value.\, \textbf{case } v \textbf{ of} \\
\qquad\quad VI\ (n : nat) \Rightarrow Num\ n \quad \text{-- abnormal branch} \\
\qquad \mid VF\ (fun : Value \to Value) \Rightarrow \\
\qquad\quad Lam\ (\lambda e' : Exp.\, (reify'\ f'\ (fun\ (eval'\ f'\ e'))))) \\
\textbf{in let } eval : Exp \to Value = eval'\ f \textbf{ in}
\end{array}
$$

Figure 1. An evaluator for the HOAS-encoded lambda calculus.

The definition of the evaluator is quite straightforward, although it is worth noting that, because λC_{β} has yet have exception mechanism, we have to pattern match on all possibilities. (That is why we have *abnormal* branches in the above code.) Thanks to the flexibility of the μ primitive, mutual recursion can be encoded by using records!

Evaluation of a lambda expression proceeds as follows:

$$
\begin{array}{l}
\textbf{let } test : Exp = App\ (Lam\ (\lambda x : Exp.\, x))\ (Num\ 42) \\
\textbf{in } show\ (eval\ test) \quad \text{-- return 42}
\end{array}
$$

Nested datatypes 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:

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

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

$$
\textbf{data } B\ (a : \star) = One\ a \mid Two\ (B\ (PairT\ a\ a));
$$

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:

$$
\begin{array}{l}
\textbf{let } B : \star \to \star = \mu\, X : \star \to \star. \\
\quad \lambda a : \star.\, (B : \star) \to (a \to B) \to (X\ (PairT\ a\ a) \to B) \to B \\
\textbf{in}
\end{array}
$$

Because of the polymorphic recursive type ($\mu X : \star \to \star$) 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 2.

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 pairs : (a : ⋆) → List a → List (PairT a a) =
  μ pairs′ : (a : ⋆) → List a → List (PairT a a).
    λa : ⋆. λxs : List a.
      case xs of
        Nil ⇒ Nil (PairT a a)
      | Cons (y : a) (ys : List a) ⇒
          case ys of Nil ⇒
            Nil (PairT a a)
          | Cons (y′ : a) (ys′ : List a) ⇒
            Cons (PairT a a) (P a a y y′) (pairs′ a ys′)
in
let fromList : (a : ⋆) → List a → B a =
  μ from′ : (a : ⋆) → List a → B a.
    λa : ⋆. λxs : List a.
      case xs of
        Nil ⇒ Two a (from′ (PairT a a) (pairs a (Nil a)))
      | Cons (x : a) (xs′ : List a) ⇒
        case xs′ of
          Nil ⇒ One a x
        | Cons (y : a) (zs : List a) ⇒
            Two a (from′ (PairT a a) (pairs a xs))
in
```

Figure 2. Construct a perfect binary tree from a list

```
let twos : (a : ⋆) → B a → nat =
  μ twos′ : (a : ⋆) → B a → nat.
    λa : ⋆. λx : B a.
      case x of
        One (y : a) ⇒ 0
      | Two (c : B (PairT a a)) ⇒
        1 + twos′ (PairT a a) c
in
let powerTwo : Nat → nat =
  λn : 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, type system and meta-theory of λC_β.

4.1 Syntax

Figure 3 shows the syntax of λC_β, including expressions, contexts and values. The syntax follows λC and straightforward to understand, while there are 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 $\Box$ representing the type of *types* and *sorts* respectively, and an axiom $\star : \Box$ 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 $\tau_2 : \Box$, 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 $\Pi\, x : \Box.x \to x$ is not allowed in λC, because $\Box$ 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 $(x : \tau_1) \to \tau_2$ of the product for clarity. By convention, we also use the syntactic sugar $\tau_1 \longrightarrow \tau_2$ 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 $\mathsf{cast}^\uparrow$ and $\mathsf{cast}_\downarrow$ (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: $\mathsf{cast}_\downarrow$ stands for the β-reduction of types, while $\mathsf{cast}^\uparrow$ is the inverse (see examples in §**??**). $\mathsf{cast}^\uparrow$ and $\mathsf{cast}_\downarrow$ 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 4 shows the *call-by-name* operational semantics, defined by one-step reduction relations. Three base cases include S_Beta for β-reduction, S_Mu for recursion unrolling and S_CastDownUp for cast canceling. Two inductive case, S_App and S_CastDown, define reduction in the head position of an application, and in the $\mathsf{cast}_\downarrow$ inner expression respectively.

The reduction rules are *weak* in the sense that it is not allowed to reduce inside a λ-term or $\mathsf{cast}^\uparrow$-term which is viewed as a value (see Figure 3). 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

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

Figure 3. Syntax of λC_β

$\boxed{e \longrightarrow e'}$ One-step reduction

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

$$\frac{}{\mu\, x : \tau.e \longrightarrow e[x \mapsto \mu\, x : \tau.e]}\ \text{S\_Mu}$$

$$\frac{}{\mathsf{cast}_{\downarrow}\,(\mathsf{cast}^{\uparrow}\,[\tau]e) \longrightarrow e}\ \text{S\_CastDownUp}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{S\_App}$$

$$\frac{e \longrightarrow e'}{\mathsf{cast}_{\downarrow}\ e \longrightarrow \mathsf{cast}_{\downarrow}\ e'}\ \text{S\_CastDown}$$

Figure 4. Operational semantics of λC_β

is equivalent to a recursive function that should be allowed to unroll without restriction.

Expression typing Figure 5 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 T_Pi and T_Mu check the type well-formedness of the dependent function and polymorphic recursion. Rule T_Mu indicates that the polymorphic recursion $\mu\, x : \tau.e$ should have the same type τ as the binder x and the body e.

Rules T_CastUp and T_CastDown deal with the explicit type conversion. They differ from the conventional implicit type conversion rule (see **??**) in λC: cast rules are syntax-directed and only perform *one-step* type conversion for every cast operator. Specifically speaking, if given $\Gamma \vdash e : \tau_2$ and $\tau_1 \longrightarrow \tau_2 \longrightarrow \tau_3$, then $\mathsf{cast}^{\uparrow}\,[\tau_1]e$ expands the type of e from τ_2 to τ_1, while $\mathsf{cast}_{\downarrow}\ e$ reduces the type of e from τ_2 to τ_3.

Giving up the implicit conversion rule means that we do not have β-equality but only have *syntactical equality* (or α-equality) for types in λC_β. By limiting the reduction relation to only one step, evaluating type-level recursive terms is fully controlled by the cast operators: the number of evaluation steps of is equal to the number of cast operators, which is finite. Thus, this prevents the type checker from looping forever when evaluating recursive terms.

Finally, we look back at the polymorphic recursion $\mu\, x : \tau.e$ again. If it appears on the term level, the reduction rule S_Mu makes the μ-operator behave like a fixpoint. Because evaluating $\mu\, x : \tau.e$ obtains its unrolling $e[x \mapsto \mu\, x : \tau.e]$, i.e. substituting all x in e with the whole μ-term itself, just like a fixpoint. If the recursion appears on the type level, assume there exist expressions e_1 and e_2 such that

$$\begin{aligned} e_1 &: \mu\, x : \tau.e \\ e_2 &: e[x \mapsto \mu\, x : \tau.e] \end{aligned}$$

Note that e_1 and e_2 have distinct types but the type of e_2 is the unrolling of e_1's type, which follows the one-step reduction relation by rule S_Mu:

$$\mu\, x : \tau.e \longrightarrow e[x \mapsto \mu\, x : \tau.e]$$

Thus, by applying rules T_CastUp and T_CastDown, we can obtain the following typing results:

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

Thus, $\mathsf{cast}^{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ witness the isomorphism between the original recursive type and its unrolling, which behave the same as fold and unfold operations in iso-recursive types.

4.3 Meta-theory

We now discuss the meta-theory of λC_β. We focus on two properties: the decidability of type checking and the type-safety of the language. First, we want to show type checking λC_β is decidable without normalizing property. The type checker will not be stuck by type-level non-termination. Second, the core language is type safe, proven by standard subject reduction and progress lemmas.

Decidability of type checking For the decidability, we need to show there exists a type checking algorithm, which returns a unique type for a well-formed expression and never loops forever. For most expression typing rules, it is straightforward to follow the syntax-directed judgement to derive a decidable type checking procedure. For rules T_CastUp and T_CastDown, one of premises needs to judge if two types τ_1 and τ_2 holds follows the one-step reduction relation, i.e. check if $\tau_1 \longrightarrow \tau_2$ holds.

We need to show checking such relation is decidable. An intuitive method is to reduce the type τ_1 by one step to obtain τ_1', and compare if τ_1' and τ_2 are syntactically equal. Such method is decidable only if such τ_1' is *unique*, i.e. reducing τ_1 will get only a single result. Otherwise, assuming $e : \tau_1$ with $\tau_1 \longrightarrow \sigma_1$ and $\tau_1 \longrightarrow \sigma_2$ holding at the same time, the type of $\mathsf{cast}_{\downarrow}\ e$ can be either σ_1 or σ_2, which is not decidable. Thus, we have to show the uniqueness one-step reduction:

Lemma 4.1 (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. □

With this result, checking if $\tau_1 \longrightarrow \tau_2$ holds is decidable and rules T_CastUp and T_CastDown are therefore decidable. We can finally conclude the decidability of type checking:

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

$$\frac{}{\vdash \varnothing}\ \textsc{Env\_Empty}$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau : \star}{\vdash \Gamma, x : \tau}\ \textsc{Env\_Var}$$

$\boxed{\Gamma \vdash e : \tau}$ Expression typing

$$\frac{}{\varnothing \vdash \star : \star}\ \textsc{T\_Ax}$$

$$\frac{\vdash \Gamma \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{T\_Var}$$

$$\frac{\Gamma \vdash e_1 : (\Pi\, x : \tau_2.\tau_1) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1[x \mapsto e_2]}\ \textsc{T\_App}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash (\Pi\, x : \tau_1.\tau_2) : \star}{\Gamma \vdash (\lambda x : \tau_1.e) : (\Pi\, x : \tau_1.\tau_2)}\ \textsc{T\_Lam}$$

$$\frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma, x : \tau_1 \vdash \tau_2 : \star}{\Gamma \vdash (\Pi\, x : \tau_1.\tau_2) : \star}\ \textsc{T\_Pi}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau \qquad \Gamma \vdash \tau : \star}{\Gamma \vdash (\mu\, x : \tau.e) : \tau}\ \textsc{T\_Mu}$$

$$\frac{\Gamma \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 : \star \qquad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\mathsf{cast}^{\uparrow}\,[\tau_1]e) : \tau_1}\ \textsc{T\_CastUp}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_2 : \star \qquad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\mathsf{cast}_{\downarrow}\ e) : \tau_2}\ \textsc{T\_CastDown}$$

Figure 5. Typing rules of λC_β

Lemma 4.2 (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. □

Note that when proving the decidability of type checking, we do not rely on the normalizing property. Because explicit type conversion rules of λC_β use one-step reduction relation, which is already decidable according to Lemma 4.1. We do not need to further require the normalization of terms. This is different from the proof for λC which requires the language is normalizing [cite the PTS paper]. Because λC's conversion rule needs to examine the β-equivalence of terms, which is decidable only if every term has a normal form.

Type safety Proof of the type safety (or soundness) of λC_β is fairly standard by subject reduction (or preservation) and progress lemmas. The subject reduction proof relies on the substitution lemma. We give the proof sketch of related lemmas as follows:

Lemma 4.3 (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 $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$. □

Lemma 4.4 (Subject reduction)**.** *If* $\Gamma \vdash e : \sigma$ *and* $e \twoheadrightarrow e'$ *then* $\Gamma \vdash e' : \sigma$.

Proof. (Sketch) 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 $\longrightarrow$. □

Lemma 4.5 (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 $\vdash e : \sigma$. □

5. 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.

5.1 Extended Syntax

The syntax of λC_{suf} is shown in Figure 6. Compared with λC_β, λC_{suf} has a new syntax category: a program, consisting of a list of datatype declarations, followed by a 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:

$$K : \Pi\overline{u : \kappa}^n.\Pi\overline{x} : \overline{\tau} \to D\,\overline{u}^n$$

The first n quantified type variables $\overline{u}$ appear in the same order in the return type $D\,\overline{u}$. 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			
pgm	$::=$	$\overline{decl}; e$	Declarations
$decl$	$::=$	$\mathbf{data}\ D\ \overline{u : \kappa} = \overline{\mid K\ \tau}$	Datatype
Terms			
u	$::=$	$x \mid K$	Variables and constructors
$e, \tau, \sigma, \upsilon, \kappa$	$::=$	u	Term atoms
		$\ldots$	
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}\ \overline{p \Rightarrow e}$	Case analysis
p	$::=$	$K\ \overline{x : \tau}$	Pattern
Environments			
Γ	$::=$	$\varnothing$	Empty
	$\mid$	$\Gamma, u : \tau$	Variable binding

Figure 6. Syntax of λC_{suf} (e for terms; τ, σ, υ for types; κ for kinds)

With datatypes, it is easy to encode *records* as syntactic sugar of simple datatypes, as shown in Figure 7.

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

Figure 7. Syntactic sugar for records

5.2 Extended Typing Rules

The type system of λC_{suf} is shown in Figure 8. To save space, we only show the new typing rules. Furthermore, we sometimes adopt the following syntactic convention:

$$\overline{\tau}^n \to \tau_r \equiv \tau_1 \to \cdots \to \tau_n \to \tau_r$$

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 ensuing 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.

5.3 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 9 shows the translation rules, which are the typing rules in Figure 8 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 datatype.

The rest of the translation rules hold few surprises.

6. Related Work

7. Conclusion

Conclusion and related work.

Acknowledgments

Thanks to Blah. This work is supported by Blah.

A. Full specification of source language

A.1 Syntax

See Figure 10.

A.2 Expression typing

See Figure 11.

A.3 Translation to the core

See Figure 12.

B. Proofs about core language

B.1 Properties

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

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 [**?**]):

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

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

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

□

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: $\dfrac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau \qquad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau : \star}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\mu\, y : \tau.e_1) : \tau}$

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: $\dfrac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau_2 \qquad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau_1 : \star \qquad \tau_1 \longrightarrow \tau_2}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\mathsf{cast}^{\uparrow}\,[\tau_1]e_1) : \tau_1}$

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, $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}^{\uparrow}\,[\tau_1^*]{e_1}^*) : \tau_1^*$. Thus we have $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}^{\uparrow}\,[\tau_1]e_1)^* : \tau_1^*$ which is just the result.

Case T_CASTDOWN: $\dfrac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau_1 \qquad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau_2 : \star \qquad \tau_1 \longrightarrow \tau_2}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\mathsf{cast}_{\downarrow}\, e_1) : \tau_2}$

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, $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}_{\downarrow}\, {e_1}^*) : \tau_2^*$. Thus we have $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}_{\downarrow}\, e_1)^* : \tau_2^*$ which is just the result.

□

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

$\boxed{\Gamma \vdash pgm : \tau}$

(Pgm)

$$\frac{\overline{\Gamma_0 \vdash decl : \Gamma_d} \qquad \Gamma = \Gamma_0, \overline{\Gamma_d} \qquad \Gamma \vdash e : \tau}{\Gamma_0 \vdash \overline{decl}; e : \tau}$$

$\boxed{\Gamma \vdash decl : \Gamma_d}$

(Data)

$$\frac{\Gamma \vdash \overline{\kappa} \to \star : \Box \qquad \overline{\Gamma, D : \overline{\kappa} \to \star, \overline{u : \kappa} \vdash \overline{\tau} \to D\,\overline{u} : \star}}{\Gamma \vdash (\mathbf{data}\, D\,\overline{u : \kappa} = \overline{\mid K\,\overline{\tau}}) : (D : \overline{\kappa} \to \star, \overline{K : \Pi \overline{u : \kappa}.\overline{\tau} \to D\,\overline{u}})}$$

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

(Case)

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \overline{\Gamma \vdash_p p \Rightarrow e_2 : \sigma \to \tau}}{\Gamma \vdash \mathbf{case}\, e_1\, \mathbf{of}\, \overline{p \Rightarrow e_2} : \tau}$$

$\boxed{\Gamma \vdash_p p \Rightarrow e : \sigma \to \tau}$

(Alt)

$$\frac{\theta = [\overline{u := v}] \qquad K : \Pi \overline{u : \kappa}.\,\overline{\sigma} \to D\,\overline{u} \in \Gamma \qquad \Gamma, \overline{x : \theta(\sigma)} \vdash e : \tau}{\Gamma \vdash_p K\,\overline{x : \theta(\sigma)} \Rightarrow e : D\,\overline{v} \to \tau}$$

Figure 8. Typing rules of λC_{suf}

(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 $\Gamma \vdash (\mathsf{cast}^{\uparrow}\,[\tau_1]e) : \sigma$, 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 $\Gamma \vdash (\mathsf{cast}_{\downarrow}\, e) : \sigma$, 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.

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. □

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. □

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) $\mathsf{cast}^{\uparrow}\,[\tau]e$, 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 $e = \mathsf{cast}_{\downarrow}\,(\mathsf{cast}^{\uparrow}\,[\tau]e)$: 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 $e = \mathsf{cast}_{\downarrow}\, e_1$ and e_1 is not a $\mathsf{cast}^{\uparrow}$-term: If $e_1 = v$, there is no e_1' such that $e_1 \longrightarrow e_1'$. Since e_1 is not a $\mathsf{cast}^{\uparrow}$-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, $e' = \mathsf{cast}_{\downarrow}\, e_1'$ 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 $e = x$: Trivial by rule TS_VAR and τ is the unique type of x if $x : \tau \in \Gamma$.
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 $e = \mathsf{cast}^{\uparrow}\,[\tau_1]e_1$: 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 $e = \mathsf{cast}_{\downarrow}\, e_1$: From the premises of rule T_CASTDOWN, by induction hypothesis, we can derive the type of e_1 as τ_1. By

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

(Ax)
$$\frac{}{\varnothing \vdash \star : \Box \rightsquigarrow \star}$$

(Var)
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow x}$$

(App)
$$\frac{\Gamma \vdash e_1 : (\Pi x : \tau_2.\, \tau_1) \rightsquigarrow E_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow E_2}{\Gamma \vdash e_1 e_2 : \tau_1[x := e_2] \rightsquigarrow E_1 E_2}$$

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

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

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

(Fold)
$$\frac{\Gamma \vdash e : \tau_2 \rightsquigarrow E \qquad \Gamma \vdash \tau_1 : s \qquad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\mathsf{cast}^{\uparrow}[\tau_1]\, e) : \tau_1 \rightsquigarrow \mathsf{cast}^{\uparrow}[\tau_1]\, E}$$

(Unfold)
$$\frac{\Gamma \vdash e : \tau_1 \rightsquigarrow E \qquad \Gamma \vdash \tau_2 : s \qquad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\mathsf{cast}_{\downarrow}\, e) : \tau_2 \rightsquigarrow \mathsf{cast}_{\downarrow}\, E}$$

(Case)
$$\frac{\Gamma \vdash e_1 : \sigma \rightsquigarrow E_1 \qquad \overline{\Gamma \vdash_p p \Rightarrow e_2 : \sigma \to \tau \rightsquigarrow E_2}}{\Gamma \vdash \mathbf{case}\, e_1\, \mathbf{of}\, \overline{p \Rightarrow e_2} : \tau \rightsquigarrow (\mathsf{cast}_{\downarrow}\, E_1)\, \tau\, \overline{E_2}}$$

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

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

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

(Data)
$$\frac{\Gamma \vdash \overline{\kappa} \to \star : \Box \qquad \overline{\Gamma, D : \overline{\kappa} \to \star, \overline{u : \kappa} \vdash \overline{\tau} \to D\, \overline{u} : \star}}{\Gamma \vdash (\mathbf{data}\, D\, \overline{u : \kappa} = \overline{\mid K\, \overline{\tau}}) : (D : \overline{\kappa} \to \star, \overline{K : \Pi \overline{u : \kappa}.\overline{\tau} \to D\, \overline{u}}) \rightsquigarrow E}$$

$$\begin{array}{rcl} E & ::= & \mathbf{let}\, D : \overline{\kappa} \to \star = \lambda \overline{u : \kappa}.\, \mu X : \star.\, \Pi b : \star.\, \overline{(\overline{\tau}[D\, \overline{u} := X] \to b)} \to b\, \mathbf{in} \\ & & \mathbf{let}\, K_i : \Pi \overline{u : \kappa}.\overline{\tau} \to D\, \overline{u} = \lambda \overline{(u : \kappa)}.\lambda \overline{(x : \tau)}. \\ & & \quad \mathsf{cast}^{\uparrow}[D\, \overline{u}]\, (\lambda (b : \star) \overline{(c : \overline{\tau} \to b)}.c_i\, \overline{x})\, \mathbf{in} \end{array}$$

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

(Pgm)
$$\frac{\overline{\Gamma_0 \vdash decl : \Gamma_d \rightsquigarrow E_1} \qquad \Gamma = \Gamma_0, \overline{\Gamma_d} \qquad \Gamma \vdash e : \tau \rightsquigarrow E}{\Gamma_0 \vdash \overline{decl}; e : \tau \rightsquigarrow \overline{E_1} \oplus E}$$

Figure 9. Type-directed translation from λC_{suf} to λC_{β}

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 $\twoheadrightarrow$ is the transitive and reflexive closure of $\longrightarrow$.*

Lemma B.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 $\dfrac{}{(\lambda x : \tau.e_1)\, e_2 \longrightarrow e_1[x \mapsto e_2]}$ **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 B.3(2), there exist expressions τ_1' and τ_2 such that

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

By Lemma B.3(3), the judgement (1) implies that there exists an expression τ_2' such that

$$\begin{aligned} &\Pi\, x : \tau_1'.\tau_2 \equiv \Pi\, x : \tau_1.\tau_2' \qquad (2)\\ &\Gamma, x : \tau_1 \vdash e_1 : \tau_2' \end{aligned}$$

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 $\dfrac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}$ **S_APP:**

pgm	::=		Program
	$\mid$	$\overline{decl}; E$	Declarations
$decl$	::=		Declarations
	$\mid$	$\mathsf{data}\ D\overline{u : \kappa} = \overline{K\overline{T}}$	Datatype declaration
$u,\ v$	::=		Atoms
	$\mid$	x	Variable
	$\mid$	K	Data constructor
p	::=		Patterns
	$\mid$	$K\overline{x : T}$	Pattern
$E,\ T,\ S,\ \kappa$	::=		Expressions
	$\mid$	u	Atom
	$\mid$	$\star$	Type of types
	$\mid$	$E_1\ E_2$	Application
	$\mid$	$\lambda x : T.E$	Abstraction
	$\mid$	$\Pi\, x : T_1.T_2$	Product
	$\mid$	$\mu\, x : T.E$	General recursion
	$\mid$	$\mathsf{case}\ E_1\ \mathsf{of}\ \overline{p \Rightarrow E_2}$	Case analysis
	$\mid$	$\mathsf{let}\ x : T = E_2\ \mathsf{in}\ E_1$	Let binding
Γ	::=		Contexts
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Gamma, u : T$	Atom binding

Syntactic Sugar

$T_1 \to T_2$	$\triangleq$	$\Pi\, x : T_1.T_2$	$x \notin \mathsf{FV}(T_2)$	Function type
$(x : T_1) \to T_2$	$\triangleq$	$\Pi\, x : T_1.T_2$	$x \in \mathsf{FV}(T_2)$	Dependent function type
$\mathsf{data}\ R\overline{u : \kappa} = K\{\overline{N : T}\}$	$\triangleq$	$\mathsf{data}\ R\overline{u : \kappa} = K\overline{T};$		Record
		$\mathsf{let}\ N_i : (\overline{u : \kappa}) \to R\overline{u} \to T_i =$	$\forall i$	
		$\lambda\overline{u : \kappa}.\lambda y : R\,\overline{u}.\mathsf{case}\ y\ \mathsf{of}\ K\overline{x : T} \Rightarrow x_i\ \mathsf{in}$		

Figure 10. Syntax of source language

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

$$\Gamma \vdash e_1 : (\Pi\, x : \tau_1.\tau_2)$$
$$\Gamma \vdash e_2 : \tau_1$$
$$\sigma \equiv \tau_2[x \mapsto e_2]$$

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 $\dfrac{e \longrightarrow e'}{\mathsf{cast}_\downarrow\ e \longrightarrow \mathsf{cast}_\downarrow\ e'}$ **S-CastDown:**

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

$$\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_2 : \star$$
$$\tau_1 \longrightarrow \tau_2 \qquad \sigma \equiv \tau_2$$

By induction hypothesis, we have $\Gamma \vdash e' : \tau_1$. By rule T-CastDown, we obtain $\Gamma \vdash \mathsf{cast}_\downarrow\ e' : \tau_2$. Therefore, $\sigma' \equiv \tau_2 \equiv \sigma$.

Case $\dfrac{}{\mathsf{cast}_\downarrow\ (\mathsf{cast}^\uparrow\ [\tau]e) \longrightarrow e}$ **S-CastDownUp:**

Suppose $\Gamma \vdash \mathsf{cast}_\downarrow\ (\mathsf{cast}^\uparrow\ [\tau_1]e) : \sigma$ and $\Gamma \vdash e : \sigma'$. By Lemma B.3(7), there exist expressions τ_1', τ_2 such that

$$\Gamma \vdash (\mathsf{cast}^\uparrow\ [\tau_1]e) : \tau_1' \tag{3}$$
$$\tau_1' \longrightarrow \tau_2 \tag{4}$$
$$\sigma \equiv \tau_2 \tag{5}$$

By Lemma B.3(6), the judgement (3) implies that there exists an expression τ_2' such that

$$\Gamma \vdash e : \tau_2' \tag{6}$$
$$\tau_1 \longrightarrow \tau_2' \tag{7}$$
$$\tau_1' \equiv \tau_1 \tag{8}$$

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 $\dfrac{}{\mu\, x : \tau.e \longrightarrow e[x \mapsto \mu\, x : \tau.e]}$ **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 $x \notin \mathsf{FV}(\tau)$ holds implicitly. Hence, by the definition of substitution, we obtain $\tau[x \mapsto \mu\, x : \tau.e] \equiv \tau$. Therefore, $\sigma' \equiv \tau[x \mapsto \mu\, x : \tau.e] \equiv \tau \equiv \sigma$.

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

$$\frac{}{\vdash \varnothing}\ \textsc{Ctx\_Empty}$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash T : \star}{\vdash \Gamma, x : T}\ \textsc{Ctx\_Var}$$

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

$$\frac{\overline{\Gamma_0 \vdash decl : \Gamma'} \qquad \Gamma = \Gamma_0, \overline{\Gamma'} \qquad \Gamma \vdash E : T}{\Gamma_0 \vdash (\overline{decl}; E) : T}\ \textsc{Tpgm\_Pgm}$$

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

$$\frac{\Gamma \vdash (\overline{u : \kappa}) \to \star : \star \qquad \overline{\Gamma, D : (\overline{u : \kappa}) \to \star, \overline{u : \kappa} \vdash \overline{T} \to D\overline{u} : \star}}{\Gamma \vdash (\mathsf{data}\ D\overline{u : \kappa} = \overline{K\overline{T}}) : (D : (\overline{u : \kappa}) \to \star, \overline{K : (\overline{u : \kappa}) \to \overline{T} \to D\overline{u}})}\ \textsc{Tdecl\_Data}$$

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

$$\frac{K : (\overline{u : \kappa}) \to \overline{S} \to D\overline{u} \in \Gamma \qquad \Gamma, \overline{x : S[\overline{u \mapsto v}]} \vdash E : T \qquad \Gamma \vdash S[\overline{u \mapsto v}] : \star}{\Gamma \vdash K\overline{x : S[\overline{u \mapsto v}]} \Rightarrow E : D\overline{v} \to T}\ \textsc{Tpat\_Alt}$$

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

$$\frac{}{\varnothing \vdash \star : \star}\ \textsc{TS\_Ax}$$

$$\frac{\vdash \Gamma \qquad x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{TS\_Var}$$

$$\frac{\Gamma \vdash E_1 : (\Pi\, x : T_2.T_1) \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1\ E_2 : T_1[x \mapsto E_2]}\ \textsc{TS\_App}$$

$$\frac{\Gamma, x : T_1 \vdash E : T_2 \qquad \Gamma \vdash (\Pi\, x : T_1.T_2) : \star}{\Gamma \vdash (\lambda x : T_1.E) : (\Pi\, x : T_1.T_2)}\ \textsc{TS\_Lam}$$

$$\frac{\Gamma \vdash T_1 : \star \qquad \Gamma, x : T_1 \vdash T_2 : \star}{\Gamma \vdash (\Pi\, x : T_1.T_2) : \star}\ \textsc{TS\_Pi}$$

$$\frac{\Gamma, x : T \vdash E : T \qquad \Gamma \vdash T : \star}{\Gamma \vdash (\mu\, x : T.E) : T}\ \textsc{TS\_Mu}$$

$$\frac{\Gamma \vdash E_1 : S \qquad \overline{\Gamma \vdash p \Rightarrow E_2 : S \to T} \qquad \overline{\Gamma \vdash S \to T : \star}}{\Gamma \vdash \mathsf{case}\ E_1\ \mathsf{of}\ \overline{p \Rightarrow E_2} : T}\ \textsc{TS\_Case}$$

$$\frac{\Gamma \vdash E_2 : T \qquad \Gamma \vdash E_1[x \mapsto E_2] : S}{\Gamma \vdash \mathsf{let}\ x : T = E_2\ \mathsf{in}\ E_1 : S}\ \textsc{TS\_Let}$$

Figure 11. Typing rules of source language

□

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) $\mathsf{cast}^{\uparrow}\,[\tau]e$.

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 $e = \mathsf{cast}_{\downarrow}\ e_1$**:** 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 $\mathsf{cast}^{\uparrow}$-term such that $e_1 \equiv \mathsf{cast}^{\uparrow}\,[\tau_1]e_1'$ for some e_1' satisfying $\vdash$

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

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

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

$$\frac{\Gamma \vdash (\overline{u : \kappa}) \to \star : \star \rightsquigarrow (\overline{u : \sigma}) \to \star \qquad \overline{\Gamma, D : (\overline{u : \kappa}) \to \star, \overline{u : \kappa} \vdash \overline{T} \to D\overline{u} : \star \rightsquigarrow \overline{\tau} \to D\overline{u}}}{\Gamma \vdash (\mathsf{data}\ D\overline{u : \kappa} = \overline{K\overline{T}}) : (D : (\overline{u : \kappa}) \to \star, \overline{K : (\overline{u : \kappa}) \to \overline{T} \to D\overline{u}}) \rightsquigarrow e} \text{ TR}_{\text{DECL\_DATA}}$$

$$e \triangleq \mathsf{let}\ D : (\overline{u : \sigma}) \to \star = \mu\, X : (\overline{u : \sigma}) \to \star.\lambda\overline{u : \sigma}.(\alpha : \star) \to \overline{(\overline{\tau}[D \mapsto X] \to \alpha)} \to \alpha\ \mathsf{in}$$
$$\mathsf{let}\ K_i : (\overline{u : \sigma}) \to \overline{\tau} \to D\overline{u} = \lambda\overline{u : \sigma}.\lambda\overline{x : \tau}.\mathsf{cast}^n_{\uparrow}\,[D\overline{u}](\lambda\alpha : \star.\lambda\overline{b : \overline{\tau} \to \alpha}.b_i\ \overline{x})\ \mathsf{in}$$

$\boxed{\Gamma \vdash p \Rightarrow E : S \to T \rightsquigarrow e}$ Pattern translation

$$\frac{K : (\overline{u : \kappa}) \to \overline{S} \to D\overline{u} \in \Gamma \qquad \Gamma, \overline{x : S[\overline{u \mapsto v}]} \vdash E : T \rightsquigarrow e \qquad \Gamma \vdash S[\overline{u \mapsto v}] : \star \rightsquigarrow \sigma}{\Gamma \vdash K\overline{x : S[\overline{u \mapsto v}]} \Rightarrow E : D\overline{v} \to T \rightsquigarrow \lambda\overline{x : \sigma}.e} \text{ TR}_{\text{PAT\_ALT}}$$

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash E_1 : S \rightsquigarrow e_1 \qquad \overline{\Gamma \vdash p \Rightarrow E_2 : S \to T \rightsquigarrow e_2} \qquad \overline{\Gamma \vdash S \to T : \star \rightsquigarrow \sigma \to \tau}}{\Gamma \vdash \mathsf{case}\ E_1\ \mathsf{of}\ \overline{p \Rightarrow E_2} : T \rightsquigarrow (\mathsf{cast}^n_{\downarrow}\ e_1)\ \tau\ \overline{e_2}} \text{ TR\_CASE}$$

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

Figure 12. Translation rules of source language

$e_1' : \tau_2$. Then by rule S_CASTDOWNUP, we can obtain $\mathsf{cast}_{\downarrow}\,(\mathsf{cast}^{\uparrow}\,[\tau_1]e_1') \longrightarrow e_1'$. 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 $\mathsf{cast}_{\downarrow}\ e_1 \longrightarrow \mathsf{cast}_{\downarrow}\ e_1'$. Thus, there exists $e' \equiv \mathsf{cast}_{\downarrow}\ e_1'$ such that $e \longrightarrow e'$.

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

□