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 langauge

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[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

[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.

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 §4.

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:

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

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.

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

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

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

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

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: $H = \mu\sigma : \star. \mathsf{Int} \rightarrow \sigma$. 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:

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

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

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

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.

[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`.

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:

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

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:

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

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:

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

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 §6.)

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

3. Applications

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

3.1 Parametric HOAS

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

$$\begin{array}{l}\textbf{data } \mathit{PLambda}\ (a : \star) = \mathit{Var}\ a \\ \quad \mid \mathit{Num}\ \mathit{nat} \\ \quad \mid \mathit{Lam}\ (a \to \mathit{PLambda}\ a) \\ \quad \mid \mathit{App}\ (\mathit{PLambda}\ a)\ (\mathit{PLambda}\ a);\end{array}$$

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 $\mathit{plambda\ value} \to \mathit{value}$. The code is presented in Figure 1.

$$\begin{array}{l}\textbf{data } \mathit{Value} = \mathit{VI}\ \mathit{nat} \\ \quad \mid \mathit{VF}\ (\mathit{Value} \to \mathit{Value}); \\ \textbf{let } \mathit{eval} : \mathit{PLambda}\ \mathit{Value} \to \mathit{Value} = \\ \quad \mu\ \mathit{ev} : \mathit{PLambda}\ \mathit{Value} \to \mathit{Value}. \\ \quad\quad \lambda e : \mathit{PLambda}\ \mathit{Value}.\, \textbf{case } e \textbf{ of} \\ \quad\quad\quad \mathit{Var}\ (v : \mathit{Value}) \Rightarrow v \\ \quad\quad \mid \mathit{Num}\ (n : \mathit{nat}) \Rightarrow \mathit{VI}\ n \\ \quad\quad \mid \mathit{Lam}\ (f : \mathit{Value} \to \mathit{PLambda}\ \mathit{Value}) \Rightarrow \\ \quad\quad\quad \mathit{VF}\ (\lambda x : \mathit{Value}.\, \mathit{ev}\ (f\ x)) \\ \quad\quad \mid \mathit{App}\ (a : \mathit{PLambda}\ \mathit{Value})\ (b : \mathit{PLambda}\ \mathit{Value}) \Rightarrow \\ \quad\quad\quad \textbf{case } (\mathit{ev}\ a) \textbf{ of} \\ \quad\quad\quad\quad \mathit{VI}\ (n : \mathit{nat}) \Rightarrow \mathit{VI}\ n \quad \text{-- impossible to reach} \\ \quad\quad\quad \mid \mathit{VF}\ (f : \mathit{Value} \to \mathit{Value}) \Rightarrow f\ (\mathit{ev}\ b) \\ \textbf{in}\end{array}$$

Figure 1. Lambda Calculus in PHAOS

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

$$\begin{array}{l}\textbf{let } \mathit{show} : \mathit{Value} \to \mathit{nat} = \\ \quad \lambda e : \mathit{Value}.\, \textbf{case } e \textbf{ of} \\ \quad\quad \mathit{VI}\ (n : \mathit{nat}) \Rightarrow n \\ \quad \mid \mathit{VF}\ (f : \mathit{Value} \to \mathit{Value}) \Rightarrow 10000 \quad \text{-- impossible to reach} \\ \textbf{in} \\ \textbf{let } \mathit{example} : \mathit{PLambda}\ \mathit{Value} = \\ \quad \mathit{App}\ \mathit{Value} \\ \quad\quad (\mathit{Lam}\ \mathit{Value}\ (\lambda x : \mathit{Value}.\, \mathit{Var}\ \mathit{Value}\ X)) \\ \quad\quad (\mathit{Num}\ \mathit{Value}\ 42) \\ \textbf{in } \mathit{show}\ (\mathit{eval}\ \mathit{example}) \quad \text{-- return 42}\end{array}$$

Figure 2. Example of using PHOAS

3.2 Perfect binary trees

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

First we define a pair datatype as follows:

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

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:

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

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 3.

$$\begin{array}{l}
\textbf{let}\ pairs : (a:\star)\to List\ a\to List\ (PairT\ a\ a) =\\
\quad \mu\ pairs' : (a:\star)\to List\ a\to List\ (PairT\ a\ a).\\
\quad\quad \lambda a:\star.\,\lambda xs : List\ a.\\
\quad\quad\quad \textbf{case}\ xs\ \textbf{of}\\
\quad\quad\quad\quad Nil \Rightarrow Nil\ (PairT\ a\ a)\\
\quad\quad\quad\quad \mid Cons\ (y:a)\ (ys:List\ a)\Rightarrow\\
\quad\quad\quad\quad\quad \textbf{case}\ ys\ \textbf{of}\ Nil\Rightarrow\\
\quad\quad\quad\quad\quad\quad Nil\ (PairT\ a\ a)\\
\quad\quad\quad\quad\quad \mid Cons\ (y':a)\ (ys':List\ a)\Rightarrow\\
\quad\quad\quad\quad\quad\quad Cons\ (PairT\ a\ a)\ (Pair\ a\ a\ y\ y')\ (pairs'\ a\ ys')\\
\textbf{in}\\
\textbf{let}\ fromList : (a:\star)\to List\ a\to B\ a =\\
\quad \mu\ from' : (a:\star)\to List\ a\to B\ a.\\
\quad\quad \lambda a:\star.\,\lambda xs : List\ a.\\
\quad\quad\quad \textbf{case}\ xs\ \textbf{of}\\
\quad\quad\quad\quad Nil\Rightarrow Two\ a\ (from'\ (PairT\ a\ a)\ (pairs\ a\ (Nil\ a)))\\
\quad\quad\quad\quad \mid Cons\ (x:a)\ (xs':List\ a)\Rightarrow\\
\quad\quad\quad\quad\quad \textbf{case}\ xs'\ \textbf{of}\\
\quad\quad\quad\quad\quad\quad Nil\Rightarrow One\ a\ x\\
\quad\quad\quad\quad\quad\quad \mid Cons\ (y:a)\ (zs:List\ a)\Rightarrow\\
\quad\quad\quad\quad\quad\quad\quad Two\ a\ (from'\ (PairT\ a\ a)\ (pairs\ a\ xs))\\
\textbf{in}
\end{array}$$

Figure 3. Construct a perfect binary tree from a list

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

$$\begin{array}{l}
\textbf{let}\ twos : (a:\star)\to B\ a\to nat =\\
\quad \mu\ twos' : (a:\star)\to B\ a\to nat.\\
\quad\quad \lambda a:\star.\,\lambda x : B\ a.\\
\quad\quad\quad \textbf{case}\ x\ \textbf{of}\\
\quad\quad\quad\quad One\ (y:a)\Rightarrow 0\\
\quad\quad\quad\quad \mid Two\ (c:B\ (PairT\ a\ a))\Rightarrow\\
\quad\quad\quad\quad\quad 1 + twos'\ (PairT\ a\ a)\ c\\
\textbf{in}\\
\textbf{let}\ powerTwo : Nat\to nat =\\
\quad \lambda n : Nat.\, twos\ nat\ (fromList\ nat\ (take\ n\ (repeat\ 1)))\\
\textbf{in}\ powerTwo\ (S\ (S\ (S\ (S\ Z))))\quad \text{-- return 2}
\end{array}$$

4. Explicit Calculus of Constructions with Recursion

In this section, we present our core language, the explicit Calculus of Constructions with recursion (λC_β). Based on the Calculus of Constructions (λC), λC_β 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 μ-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 λC when evaluating recursive types. In λC_β, 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 λC_β can be decidable without requiring strong normalization.

4.1 Syntax

The basic syntax of λC_β is shown in Figure 4, including abstract syntax of expressions, contexts and values. Inherited from λC, λC_β uses a single syntactic level to represent terms, types and kinds, while other typed intermediate languages based on System F or F_ω usually distinguish them, e.g. System F_c of GHC, or μF^* 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 τ 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 τ as a type-level expression. For example, in $\sigma : \star$, the term-level expression σ is traditionally a type with kind $\star$.

Similar to λC, λC_β uses a product form $\Pi\, x : \tau_1 . \tau_2$ to represent both traditional and dependent function types. We interchangeably use the arrow form $(x : \tau_1) \to \tau_2$ of the product in the source language for brevity. 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.

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

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 λC 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 λC_{exp} contains typing judgements and operational semantics. Figure 5 lists operational semantics for λC_{exp} that defines rules for one-step reduction, including the β-reduction rule and $\mathsf{cast}_{\downarrow}$ rules. The expressions will be reduced by applying rules one or more times. Rule S_CASTDOWN prevents the reduction from stalling with $\mathsf{cast}_{\downarrow}$ and continues to reduce the inner expression. Rule S_CASTDOWNUP states that $\mathsf{cast}_{\downarrow}$ cancels the $\mathsf{cast}^{\uparrow}$ of an expression.

Figure 6 lists the typing judgements to check the validity of expressions. Most rules are straightforward and similar with the ones in λC. For example, rule T_AX states that the "type" of sort $\star$ is a kind. This is derived from an axiom in λC, that the highest sort is $\square$, making the type system predicative. Rule T_PI

$$
\begin{array}{llll}
e,\ \tau,\ \sigma & ::= & & \text{Expressions} \\
& | & x & \text{Variable} \\
& | & \star & \text{Type of types} \\
& | & e_1\ e_2 & \text{Application} \\
& | & \lambda x : \tau . e & \text{Abstraction} \\
& | & \Pi\, x : \tau_1 . \tau_2 & \text{Product} \\
& | & \mathsf{cast}^{\uparrow}\, [\tau] e & \text{Cast up to type} \\
& | & \mathsf{cast}_{\downarrow}\ e & \text{Cast down by reduction} \\
& | & \mu\, x : \tau . e & \text{General recursion} \\
\\
\Gamma & ::= & & \text{Contexts} \\
& | & \varnothing & \text{Empty} \\
& | & \Gamma, x : \tau & \text{Variable binding} \\
\\
v & ::= & & \text{Values} \\
& | & \lambda x : \tau . e & \text{Abstraction} \\
& | & \Pi\, x : \tau_1 . \tau_2 & \text{Product} \\
& | & \mathsf{cast}^{\uparrow}\, [\tau] e & \text{Cast up}
\end{array}
$$

Figure 4. Syntax of λC_β

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

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

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

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

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

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

Figure 5. Operational semantics of λC_{exp}

allows us to type dependent products. There are four possible combinations of types of τ_1 and τ_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 λC for typing rules of λC_{exp} is that rule T_CastUp and T_CastDown are added to check the type conversion primitives $\mathsf{cast}^{\uparrow}$ and $\mathsf{cast}_{\downarrow}$, and the implicit type conversion rule of λC is removed, which is the rule as follows:

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

This rule is necessary for λC because of the premise requirements of the application rule T_App:

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

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

- e_2 can be an arbitrary term so its type τ_2 is not necessary in normal form which might break the type checking of e_1, e.g. suppose $e_1 : \sigma \to \tau$ and $e_2 : \tau_2$, where τ_2 is an application $(\lambda x : \star . x)\ \sigma$. By Tcc_Conv, $(\lambda x : \star . x)\ \sigma$ is β-equivalent to σ, 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 β-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 $e_1 : \sigma \to \tau$ and $e_2 : (\lambda x : \star . x)\ \sigma$, we do the application by term $e_1\ (\mathsf{cast}_{\downarrow}\ e_2)$. Since $(\lambda x : \star . x)\ \sigma \longrightarrow \sigma$, $\mathsf{cast}_{\downarrow}\ e_2 : \sigma$, the term $e_1\ (\mathsf{cast}_{\downarrow}\ e_2)$ 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 $e_1\ (\mathsf{cast}^{\uparrow}\, [(\lambda x : \star . x)\ \tau_2] e_2)$. Noting that $(\lambda x : \star . x)\ \tau_2 \longrightarrow \tau_2$, the term conforms to rule T_CastUp. Thus $\mathsf{cast}^{\uparrow}\, [(\lambda x : \star . x)\ \tau_2] e_2 : ((\lambda x : \star . x)\ \tau_2)$ and the term $e_1\ (\mathsf{cast}^{\uparrow}\, [(\lambda x : \star . x)\ \tau_2] e_2)$ can be type-checked by the rule T_App.

Therefore, it is feasible to replace implicit conversion rules of λC with explicit type conversion rules.

$\boxed{\Gamma \vdash e : \tau}$ Typing rules of λC

$$\frac{}{\varnothing \vdash \star : \square}\ \textsc{Tcc\_Ax}$$

$$\frac{\Gamma \vdash \tau : s \qquad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Tcc\_Var}$$

$$\frac{\Gamma \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 : s \qquad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x : \tau_1 \vdash e : \tau_2}\ \textsc{Tcc\_Weak}$$

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

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

$$\frac{\Gamma \vdash \tau_1 : s \qquad \Gamma, x : \tau_1 \vdash \tau_2 : t}{\Gamma \vdash (\Pi\, x : \tau_1 . \tau_2) : t}\ \textsc{Tcc\_Pi}$$

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

Figure 6. Typing rules of λC

4.3 Decidability and soundness without strong normalization

The conversion rule of λC is not syntax-directed because it can be implicitly applied at any time in a derivation. The β-equality premise of the rule also leads to the decidability of type checking relying on the strong normalization property of λC. Suppose strong normalization does not hold in the type system, then we can find a type τ_1 such that there exists at least one reduction sequence which does not terminate. Notice that any type τ_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 λC, 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 λC_{exp})**.** *Let Γ be an environment, e and τ be expressions of λC_{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. □

Notice that new explicit type conversion rules are syntax-directed and do not include the β-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 λC_{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 λC_{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 $e \longrightarrow e'$ then $\Gamma \vdash e' : \tau$.*

Proof. By induction on rules in Figure 5. □

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

Proof. By induction on rules in Figure 6. □

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 λC_{exp} does not rely on strong normalization for decidable type checking and soundness. Thus it is safe to combine general recursion with λC_{exp} under the control of explicit type conversion operations $\mathsf{cast}^{\uparrow}$ and $\mathsf{cast}_{\downarrow}$. We extend λC_{exp} into λC_{β} by introducing one unified primitive called μ-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 μ-notation

Based on the syntax of λC_{exp}, we add the following μ-notation for λC_{β} (the same part as λC_{exp} is left out):

$$
\begin{array}{llll}
e,\ \tau & ::= & & \text{Expressions} \\
 & | & \ldots & \\
 & | & \mu\, x : \tau . e & \text{General recursion}
\end{array}
$$

The μ-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 λC_{exp} that terms and types are not distinguished.

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

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

And the one-step reduction rule is as follows:

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

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 β-equivalence. But such result cannot be directly obtained because of the removal of implicit conversion rule. Instead, by using explicit cast operations of λC_{exp}, we can obtain the following transformation between e and e':

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

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

5.2 Decidability and soundness

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

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

As what we previously illustrate in Section 4.3, the type checking of λC_{exp} can always terminate because the derivation is finite without the implicit conversion rule. With the mu-notation in λC_{β}, the decidability of type checking still holds because the type level recursion is explicitly controlled by cast operations. Notice that in the typing rule of $\mathsf{cast}^{\uparrow}$ and $\mathsf{cast}_{\downarrow}$, the reduction is performed by one step. Thus the reduction sequences are always finite. Also by adopting the definitional equality, to judge if two terms are equal in the type checking is also decidable. Therefore, the new T_Mu rule is decidable for type checking.

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

6. Surface language

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

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

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

6.1 Extended Syntax

The syntax of λC_{suf} is shown in Figure 7. Compared with λC_β, λC_{suf} has a new syntax category: a program, consisting of a list of datatype declarations, followed by 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
$decl$	$::=$	$\mathbf{data}\, D\,\overline{u : \kappa} = \overline{\mid K\,\overline{\tau}}$	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 7. 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 8.

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

Figure 8. Syntactic sugar for records

6.2 Extended Typing Rules

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

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

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.

6.3 Translation Overview

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

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

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

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

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

A. Specification of core language

A.1 Syntax

$e,\ \tau,\ \sigma$	$::=$		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 to type
	$\mid$	$\mathsf{cast}_{\downarrow}\ e$	Cast down by reduction
	$\mid$	$\mu\, x : \tau.e$	General recursion
Γ	$::=$		Contexts
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Gamma, x : \tau$	Variable binding
v	$::=$		Values
	$\mid$	$\lambda x : \tau.e$	Abstraction
	$\mid$	$\Pi\, x : \tau_1.\tau_2$	Product

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

(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 9. Typing rules of λC_{suf}

$\mid \quad \mathsf{cast}^{\uparrow}\,[\tau] e$ Cast up

A.2 Operational semantics and expression typing

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

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

$$\frac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2} \text{ S\_APP}$$

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

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

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

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

$$\frac{}{\vdash \varnothing} \text{ ENV\_EMPTY}$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau : \star}{\vdash \Gamma, x : \tau} \text{ ENV\_VAR}$$

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

$$\frac{}{\varnothing \vdash \star : \star} \text{ T\_AX}$$

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

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

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

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

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

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

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

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* $\mathsf{FV}(e) \subseteq \mathsf{dom}(\Gamma)$ *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 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: $\dfrac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau \qquad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau : \star}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\mu\, y : \tau.e_1) : \tau}$

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.

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

(Ax) $$\frac{}{\varnothing \vdash \star : \square \leadsto \star}$$

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

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

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

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

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

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

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

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

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

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

$\boxed{\Gamma \vdash \mathit{decl} : \Gamma_d \leadsto E}$

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

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

$\boxed{\Gamma \vdash \mathit{pgm} : \tau \leadsto E}$

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

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

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

(1) If $\Gamma \vdash x : \sigma$, then there exist an expression τ 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 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 a sort $\star$ and 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 there exists a sort $\star$ such that $\Gamma \vdash \tau : \star$, $\sigma \equiv \tau$ and $\Gamma, x : \tau \vdash e : \tau$.

(6) If $\Gamma \vdash (\mathsf{cast}^{\uparrow}\, [\tau_1] e) : \sigma$, then there exist an expression τ_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 $\Gamma \vdash (\mathsf{cast}_{\downarrow}\, e) : \sigma$, then there exist expressions τ_1, τ_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

$$
\begin{array}{llll}
pgm & ::= & & \text{Program} \\
& | & \overline{decl}; E & \quad \text{Declarations} \\
\\
decl & ::= & & \text{Declarations} \\
& | & \mathsf{data}\ D\overline{u:\kappa} = \overline{K\overline{T}} & \quad \text{Datatype declaration} \\
\\
u,\ v & ::= & & \text{Atoms} \\
& | & x & \quad \text{Variable} \\
& | & K & \quad \text{Data constructor} \\
\\
p & ::= & & \text{Patterns} \\
& | & K\overline{x:T} & \quad \text{Pattern} \\
\\
E,\ T,\ S,\ \kappa & ::= & & \text{Expressions} \\
& | & u & \quad \text{Atom} \\
& | & \star & \quad \text{Type of types} \\
& | & E_1\ E_2 & \quad \text{Application} \\
& | & \lambda x : T.E & \quad \text{Abstraction} \\
& | & \Pi\, x : T_1.T_2 & \quad \text{Product} \\
& | & \mathsf{cast}^{\uparrow}\,[T]E & \quad \text{Cast up to type} \\
& | & \mathsf{cast}_{\downarrow}\, E & \quad \text{Cast down by reduction} \\
& | & \mu\, x : T.E & \quad \text{General recursion} \\
& | & \mathsf{case}\ E_1\ \mathsf{of}\ \overline{p \Rightarrow E_2} & \quad \text{Case analysis} \\
& | & \mathsf{let}\ x : T = E_2\ \mathsf{in}\ E_1 & \quad \text{Let binding} \\
\\
\Gamma & ::= & & \text{Contexts} \\
& | & \varnothing & \quad \text{Empty} \\
& | & \Gamma, u : T & \quad \text{Atom binding}
\end{array}
$$

$$
\begin{array}{llll}
& & & \text{Syntactic Sugar} \\
\mathsf{let}\ x : T = E_2\ \mathsf{in}\ E_1 & \triangleq\ E_1[x \mapsto E_2] & & \quad \text{Let binding} \\
T_1 \to T_2 & \triangleq\ \Pi\, x : T_1.T_2 & x \notin \mathsf{FV}(T_2) & \quad \text{Function type} \\
(x : T_1) \to T_2 & \triangleq\ \Pi\, x : T_1.T_2 & x \in \mathsf{FV}(T_2) & \quad \text{Dependent function type} \\
\mathsf{data}\ R\overline{u:\kappa} = K\{\overline{N:T}\} & \triangleq\ \mathsf{data}\ R\overline{u:\kappa} = K\overline{T}; & & \quad \text{Record} \\
& \quad \mathsf{let}\ N_i : (\overline{u:\kappa}) \to R\overline{u} \to T_i = & \forall i & \\
& \quad \lambda\overline{u:\kappa}.\lambda y : R\,\overline{u}.\mathsf{case}\ y\ \mathsf{of}\ K\overline{x:T} \Rightarrow x_i\ \mathsf{in} & &
\end{array}
$$

Figure 11. Syntax of source language

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

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

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

$\boxed{\vdash \Gamma}$ 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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash E : T_2 \qquad \Gamma \vdash T_1 : \star \qquad T_1 \longrightarrow T_2}{\Gamma \vdash (\mathsf{cast}^{\uparrow}\,[T_1] E) : T_1}\ \textsc{TS\_CastUp}$$

$$\frac{\Gamma \vdash E : T_1 \qquad \Gamma \vdash T_2 : \star \qquad T_1 \longrightarrow T_2}{\Gamma \vdash (\mathsf{cast}_{\downarrow}\, E) : T_2}\ \textsc{TS\_CastDown}$$

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

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

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

Figure 12. Typing rules of source language

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.

□

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 13. Translation rules of source language

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

Proof. By induction on the structure of e:

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

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

Case $e = e_1\, e_2$**, or** $\lambda x : \tau_1 . e_1$**, or** $\Pi\, x : \tau_1 . \tau_2$**, or** $\mu\, x : \tau . e_1$**:** Trivial according to Lemma C.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 C.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 Lemma C.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.

□

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 $\dfrac{}{(\lambda x : \tau . e_1)\, e_2 \longrightarrow e_1[x \mapsto e_2]}$ **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 C.3(2), there exist expressions τ_1' and τ_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 τ_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 $\dfrac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}$ **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 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 C.3(7), there exist expressions τ_1, τ_2 and a sort $\star$ 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 C.3(7), there exist expressions τ_1', τ_2 such that

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

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

$$\Gamma \vdash e : \tau_2' \quad (6)$$
$$\tau_1 \longrightarrow \tau_2' \quad (7)$$
$$\tau_1' \equiv \tau_1 \quad (8)$$

By (4, 7, 8) and Lemma 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 $\dfrac{}{\mu\, x : \tau . e \longrightarrow e[x \mapsto \mu\, x : \tau . e]}$ **S_MU:**
Suppose $\Gamma \vdash (\mu\, x : \tau . e) : \sigma$ and $\Gamma \vdash e[x \mapsto \mu\, x : \tau . e] : \sigma'$. By Lemma 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 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, $\sigma' \equiv \tau[x \mapsto \mu\, x : \tau . e] \equiv \tau \equiv \sigma$.

□

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

Case $e = e_1\, e_2$**:** By Lemma C.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 C.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 C.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 C.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 $\vdash e_1' : \tau_2$. 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 $e' \equiv e_1[x \mapsto \mu\, x : \tau . e_1]$.

□