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 $\lambda_{\star\mu}$: 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 $\lambda_{\star\mu}$, 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 pos-

sible to cause non-termination in the type-checker, because that would require a program with an infinite number of casts. Therefore, 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.

Our motivation to develop $\lambda_{\star\mu}$ is to use it as a simpler alternative to existing core languages for languages such as Haskell. The paper shows how many of programming language features of Haskell, including some of the latest extensions, can be encoded in $\lambda_{\star\mu}$ 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 $\lambda_{\star\mu}$ 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 $\lambda_{\star\mu}$.

We believe $\lambda_{\star\mu}$ is particularly well-suited as a core for Haskell-like languages. In particular the treatment of type-level computation shares similar ideas with Haskell. Although Haskell's surface language provides a rich set of mechanisms to do type-level computation [], the core language lacks fundamental mechanisms todo type-level computation. In particular, like in $\lambda_{\star\mu}$, type equality is purely syntactic (modulo alpha-conversion).

In summary, the contributions 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.

2. Overview

This section informally introduces the main features of $\lambda_{\star\mu}$. In particular, this section shows how the explicit casts in $\lambda_{\star\mu}$ can be used instead of the typical conversion rule present in calculi such as the calculus of constructions. The formal details of $\lambda_{\star\mu}$ are presented in §**??**.

2.1 The Calculus of Constructions and the Conversion Rule

The calculus of constructions (λC) [5] is a powerful higher-order typed lambda calculus supporting dependent types (among various other features). A crutial 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. This rule is important to *automatically* allows terms with equivalent types to be considered type-compatible. To illustrate, let us consider a simple example. Suppose we have a built-in base type Int and

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

Here f is a simple identity function. However, the type of x ($(\lambda y : \star . y)\ \mathsf{Int}$), which is the argument of f, is interesting: it is an identity function on types, applied to an integer. Without the conversion rule, f cannot be applied to, say 3 in λC. However, given that f is actually β-convertible to $\lambda x : \mathsf{Int} . x$, the conversion rule allows the application of f to 3 by implicitly converting $\lambda x : (\lambda y : \star . y)\ \mathsf{Int} . x$ to $\lambda x : \mathsf{Int} . x$.

Decidability of Type-Checking and Strong Normalization While the conversion rule in λC brings alot of convenience, an unfortunate consequence is that it couples decidability of type-checking with strong normalization of the calculus [4]. However strong normalization does not hold with general recursion. This is simply because due to the conversion rule, any non-terminating term would force the type checker to go into an infinitely loop (by constantly applying the conversion rule without termination), thus rendering the type system undecidable. BRUNO: show simple example. Explain issue better.

2.2 An Alternative to the Conversion Rule: Explicit Casts

BRUNO: Mention somewhere that the cast rules do *one-step* reductions. In contrast to the implicit reduction rules of λC, $\lambda_{\star\mu}$ makes it explicit as to when and where to convert one type to another. Type conversions are explicit by introducing two language constructs: $\mathsf{cast}^{\uparrow}$ (beta expansion) and $\mathsf{cast}_{\downarrow}$ (beta reduction). The benefit of this approach is that decidability of type-checking no longer is coupled with strong normalization of the calculus.

Beta Expansion The first of the two type conversions $\mathsf{cast}^{\uparrow}$, allows a type conversion provided that the resulting type is a *beta expansion* of the original type of the term. Consider the same example from §2.1. In $\lambda_{\star\mu}$, $f\ 3$ is an ill-typed application. Instead we must write the application as

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

BRUNO: how to put a space before 3? The intuition is that, $\mathsf{cast}^{\uparrow}$ is actually doing a type conversion since the type of 3 is Int and $(\lambda y : \star . y)\ \mathsf{Int}$ can be reduced to Int.BRUNO: explain why this is a beta expansionBRUNO: explain why for beta expansions we need to provide the resulting type as argument.

Beta Reduction The dual operation of $\mathsf{cast}^{\uparrow}$ is $\mathsf{cast}_{\downarrow}$, which allows a type conversion provided that the resulting type is a *beta reduction* of the original type of the term. 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. BRUNO: explain why this is a reduction

2.3 Decidability without Strong Normalization

With explicit type conversion rules the decidability of type-checking no longer depends on the normalization property. A nice consequence of this is that the type system remains decidable even in the presence of non-terminating programs at type level. To illustrate, let us consider the following 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 loop, where the latter is non-terminating.

This is not the case for $\lambda_{\star\mu}$: $\lambda_{\star\mu}$ has no conversion rule, therefore the type checker would do syntactic comparison between the two terms instead of β-equality in the above example and reject the program as ill-typed. Indeed it would be impossible to type-check the program even with the use of $\mathsf{cast}^{\uparrow}$ and/or $\mathsf{cast}_{\downarrow}$: one would need to write infinite number of $\mathsf{cast}_{\downarrow}$'s to make the type checker loop forever (e.g., $(\lambda x : d\,3.x)(\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow}\ldots z))$, but it is impossible to write such program in reality.

In summary, $\lambda_{\star\mu}$ 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 Recursion and Recursive Types

BRUNO: Show how in $\lambda_{\star\mu}$ recursion and recursive types are unified. Discuss that due to this unification the sensible choice for the evaluation strategy is call-by-name.

A simple extension to $\lambda_{\star\mu}$ is to add a simple recursion construct. With such an extension, it becomes possible to write standard recursive programs at the term level. At the same time, the recursive construct can also be used to model recursive types at the type-level. Therefore, $\lambda_{\star\mu}$ differs from other programming languages in that it unifies both recursion and recursive types by the same μ primitive. With a single language construct we get two powerful features!

Recursion The μ primitive is used to define recursive functions. For example the factorial function is defined as follows:

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

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.

Recursive types In the literature on type systems, there are two approaches to recursive types. One is called *equi-recursive*, the other *iso-recursive*. The *iso-recursive* approach treats a recursive type and its unfolding as different, but isomorphic. The isomorphism between a recursive type and it's one step unfolding is witnessed by traditionally $fold$ and $unfold$ operations.BRUNO: Explain that the casts generalize fold and unfold! In $\lambda_{\star\mu}$, 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:

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

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.

Call-by-Name 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. BRUNO: Probably needs to be improved. I'll came back to this later!

2.5 Logical Inconsistency

BRUNO: Explain that the $\lambda_{\star\mu}$ is inconsistent and discuss that this is a deliberate design decision, since we want to model languages like Haskell, which are logically inconsistent as well. BRUNO: Discuss the $* : *$ rule: since we already have inconsistency, having this rule adds expressiveness and simplifies the system.

2.6 Encoding Datatypes

With the explicit type conversion rules and the μ primitive, it is easy to encode recursive datatypes and recursive functions using datatypes. While inductive datatypes can be encoded using either the Church or the Scott encoding, we adopt the Scott encoding as it encodes 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:

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

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

As can be seen, in the function type $B \to (Nat \to B) \to B$, 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:

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

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:

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

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

3. Showcase

In this section, we showcase applications, which either Haskell needs non-trivial extensions to do that, or dependently typed languages like Coq and Agda are impossible to do, whereas we can easily achieve in $\lambda_{\star\mu}$.

Conventional datatypes Conventional datatypes like natural numbers or polymorphic lists can be easily defined in $\lambda_{\star\mu}$, as in Haskell. For example, below is the definition of polymorphic lists:

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

Because $\lambda_{\star\mu}$ 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:

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

Higher-kinded types Higher-kinded types are types that take other types and produce a new type. To support higher-kinded types, languages like Haskell have to extend their existing core languages to account for kind expressions. In $\lambda_{\star\mu}$, since all syntactical constructs are in the same level, we can easily construct higher-kinded types. We show this by an example of encoding the *Functor* class:

$$\begin{array}{l}\mathbf{rcrd}\ \mathit{Functor}\ (f:\star\to\star) = \\ \quad \mathit{Func}\ \{\mathit{fmap} : (a:\star) \to (b:\star) \to f\ a \to f\ b\};\end{array}$$

A functor is just a record that has only one field *fmap*. A Functor instance of the *Maybe* datatype is simply:

$$\begin{array}{l}\mathbf{let}\ \mathit{maybeInst} : \mathit{Functor}\ \mathit{Maybe} = \\ \quad \mathit{Func}\ \mathit{Maybe}\ (\lambda a:\star.\, \lambda b:\star.\, \lambda f : a \to b.\, \lambda x : \mathit{Maybe}\ a. \\ \quad\quad \mathbf{case}\ x\ \mathbf{of} \\ \quad\quad\quad \mathit{Nothing} \Rightarrow \mathit{Nothing}\ b \\ \quad\quad\quad \mid \mathit{Just}\ (z:a) \Rightarrow \mathit{Just}\ b\ (f\ z))\end{array}$$

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 $\lambda_{\star\mu}$ is able to express HOAS in a straightforward way. We show an example of encoding a simple lambda calculus:

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

Next we define the evaluator for our lambda calculus. As noted by [6], the evaluation function needs an extra function *reify* to invert the result of evaluation. The code is presented in Figure 1.

The definition of the evaluator is quite straightforward, although it is worth noting that the evaluator is a partial function that can cause run-time errors. Thanks to the flexibility of the μ primitive, mutual recursion can be encoded by using records!

Evaluation of a lambda expression proceeds as follows:

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

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

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

Fix as a datatype The type-level *Fix* is a good example to demonstrate the expressiveness of $\lambda_{\star\mu}$. The definition is simply:

$$\mathbf{rcrd}\ \mathit{Fix}\ (f:\star\to\star) = \mathit{In}\ \{\mathit{out} : (f\ (\mathit{Fix}\ f))\};$$

The record notation also introduces the selector function: $\mathit{out} : (f:\star\to\star) \to \mathit{Fix}\ f \to f\ (\mathit{Fix}\ f)$. The *Fix* datatype is interesting in that Coq and Agda would reject this definition because the use of the higher-kinded type parameter f could be anywhere (e.g., in a negative position). However in $\lambda_{\star\mu}$, where type-level computation is explicitly controlled, we can safely use *Fix* in the program.

Given *fmap*, many recursive shcemes can be defined, for example we can have *catamorphism* or generic function fold:

$$\begin{array}{l}\mathbf{letrec}\ \mathit{cata} : (f:\star\to\star) \to (a:\star) \to \\ \qquad\qquad \mathit{Functor}\ f \to (f\ a \to a) \to \mathit{Fix}\ f \to a = \\ \quad \lambda f:\star\to\star.\, \lambda a:\star.\, \lambda m : \mathit{Functor}\ f.\, \lambda g : f\ a \to a.\, \lambda t : \mathit{Fix}\ f. \\ \quad\quad g\ (\mathit{fmap}\ f\ m\ (\mathit{Fix}\ f)\ a\ (\mathit{cata}\ f\ a\ m\ g)\ (\mathit{out}\ f\ t))\end{array}$$

Kind polymophism for datatypes In Haskell, System $F_c^{\uparrow}$ was proposed to support kind polymorphism. However it separates expressions into terms, types and kinds, which complicates both the implementation and future extensions. $\lambda_{\star\mu}$ natively allows datatype definitions to have polymorphic kinds. Here is an example, taken from [16], of a datatype that benefits from kind polymophism: higher-kinded fixpoint combinator

$$\begin{array}{l}\mathbf{data}\ \mathit{Mu}\ (k:\star)\ (f : (k\to\star) \to k \to \star)\ (a : k) = \\ \quad \mathit{Roll}\ (f\ (\mathit{Mu}\ k\ f)\ a);\end{array}$$

Mu can be used to construct polymorphic recursive types of any kind, for instance:

$$\begin{array}{l}\mathbf{data}\ \mathit{Listf}\ (f:\star\to\star)\ (a:\star) = \mathit{Nil} \mid \mathit{Cons}\ a\ (f\ a); \\ \mathbf{let}\ \mathit{List} : \star\to\star = \lambda a:\star.\, \mathit{Mu}\ \star\ \mathit{Listf}\ a\end{array}$$

Nested datatypes and polymorphic recursion A nested datatype, also known as a *non-regular* datatype, is a parametrised datatype whose definition contains different instances of the type parameters. Functions over nested datatypes usually involve polymorphic recursion. We show that $\lambda_{\star\mu}$ is capable of defining all useful functions over a nested datatype. A simple example would be the type

Pow of power trees, whose size is exactly a power of two, declared as follows:

$$\textbf{data}\ \mathit{PairT}\ (a : \star) = P\ a\ a;$$
$$\textbf{data}\ \mathit{Pow}\ (a : \star) = \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Pow}\ (\mathit{PairT}\ a));$$

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

$$\textbf{let}\ \mathit{Pow} : \star \to \star = \mu\ X : \star \to \star.$$
$$\lambda a : \star.\ (B : \star) \to (a \to B) \to (X\ (\mathit{PairT}\ a) \to B) \to B$$

Notice how the higher-kinded type variable $X : \star \to \star$ helps encoding nested datatypes. Below is a simple function *toList* that transforms a power tree into a list:

$$\begin{array}{l}\textbf{letrec}\ \mathit{toList} : (a : \star) \to \mathit{Pow}\ a \to \mathit{List}\ a = \\ \quad \lambda a : \star.\ \lambda t : \mathit{Pow}\ a.\ \textbf{case}\ t\ \textbf{of} \\ \quad\quad \mathit{Zero}\ (x : a) \Rightarrow \mathit{Cons}\ a\ x\ (\mathit{Nil}\ a) \\ \quad\quad \mid \mathit{Succ}\ (c : \mathit{Pow}\ (\mathit{PairT}\ a)) \Rightarrow \\ \quad\quad\quad \mathit{concatMap}\ (\mathit{PairT}\ a)\ a \\ \quad\quad\quad\quad (\lambda x : \mathit{PairT}\ a.\ \textbf{case}\ x\ \textbf{of} \\ \quad\quad\quad\quad\quad P\ (m : a)\ (n : a) \Rightarrow \\ \quad\quad\quad\quad\quad\quad \mathit{Cons}\ a\ m\ (\mathit{Cons}\ a\ n\ (\mathit{Nil}\ a))) \\ \quad\quad\quad\quad (\mathit{toList}\ (\mathit{PairT}\ a)\ c)\end{array}$$

4. The Core Language without Recursion

In this section, we present $\lambda_\star$ calculus, a language based on λC with explicit type-level computation. $\lambda_\star$ is a subset of the core language $\lambda_{\star\mu}$ without general recursion, which is simple enough for meta-theoretic studies. By explicitly controlling the type-level computation with cast operators, $\lambda_\star$ can achieve decidable type checking without requiring normalizing property. In rest of this section, we demonstrate the syntax, operational semantics, typing rules and meta-theory of $\lambda_\star$.

4.1 Syntax

Figure 2 shows the syntax of $\lambda_\star$, including expressions, contexts and values. The syntax follows λC and fairly straightforward while there are still some differences in $\lambda_\star$, including the 'type-in-type' axiom and cast operators.

Unified syntactic levels $\lambda_\star$ 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 $\lambda_\star$, 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*. $\lambda_\star$ 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 $\lambda_\star$ 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.

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

Figure 2. Syntax of $\lambda_\star$

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 §**??**).

Though cast primitives make the syntax verbose when type conversion is heavily used, the implementation of type checking is simplified because typing rules of $\lambda_\star$ 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 Operational Semantics

Figure 3 shows the *call-by-name* operational semantics, defined by one-step reduction. Two base cases include S_BETA for β-reduction 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 2).

To evaluate the value of a term-level expression, we apply the one-step reduction multiple times. The number of evaluation steps is not restricted, which is possible to be infinite. The multi-step reduction can be defined as follows:

Definition 4.1 (Multi-step reduction)**.** *The relation* $\twoheadrightarrow$ *is the transitive and reflexive closure of the one-step reduction* $\longrightarrow$.

But for a type-level expression, the evaluation is driven by cast operators, which is finite. For a consecutive sequence of reductions with n steps, we use the notation $\longrightarrow_n$ to denote the relation between the initial and final expressions:

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

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

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

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

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

Figure 3. Operational semantics of $\lambda_\star$

Definition 4.2 (n-step reduction)**.** *The n-step reduction is denoted by $e_0 \longrightarrow_n e_n$, if there exists a sequence of one-step reductions $e_0 \longrightarrow e_1 \longrightarrow e_2 \longrightarrow \ldots \longrightarrow e_n$, where n is a positive integer and $e_i\ (i = 0, 1, \ldots, n)$ are valid expressions.*

4.3 Typing

Figure 4 gives the *syntax-directed* typing rules of $\lambda_\star$, including 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 there is no distinction of different syntactic levels.

Most typing rules are quite standard. We write $\vdash \Gamma$ if a context Γ is well-formed. Note that there is only a single sort $\star$, we use $\Gamma \vdash \tau : \star$ 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 check the type well-formedness of the dependent function.

We focus on rules T_CastUp and T_CastDown that define the semantics of cast operators and replace the conversion rule of λC (see **??**). The relation between the original and converted type is defined by one-step reduction (see 3). Specifically speaking, if given a judgement $\Gamma \vdash e : \tau_2$ and relation $\tau_1 \longrightarrow \tau_2 \longrightarrow \tau_3$, 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.

Finally, the definition of type equality in $\lambda_\star$ differs from λC. Without λC's conversion rule, the type of a term cannot be converted freely against β-equality, unless using cast operators. Thus, types of expressions are equal only if they are syntactically equal, i.e. satisfy the α-equality.

4.4 Meta-theory

We now discuss the meta-theory of $\lambda_\star$. We focus on two properties: the decidability of type checking and the type-safety of the language. First, we want to show type checking $\lambda_\star$ is decidable without normalizing property. The type checker will not be stuck by type-level non-termination. Second, the 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 never loops forever and returns a unique type for a well-formed expression e. This is done by induction on the length of e and ranging over typing rules. Most expression typing rules, which have only typing judgements in premises, are already decidable by induction hypothesis. Thus, it is straightforward to follow the syntax-directed judgement to derive a unique type checking result.

The critical case is for rules T_CastUp and T_CastDown. Both rules contain a premise that needs to judge if two types τ_1 and

$\boxed{\vdash \Gamma}$ 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

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

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

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

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

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

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

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

Figure 4. Typing rules of $\lambda_\star$

τ_2 follows the one-step reduction, i.e. if $\tau_1 \longrightarrow \tau_2$ holds. We need to show such τ_2 is *unique* with respect to the one-step reduction, or equivalently, reducing τ_1 by one step will get only a sole result τ_2. Otherwise, assume $e : \tau_1$ and there exists τ_2' such that $\tau_1 \longrightarrow \tau_2$ and $\tau_1 \longrightarrow \tau_2'$. Then the type of $\mathsf{cast}_{\downarrow}\ e$ can be either τ_2 or τ_2' by rule T_CastDown, which is not decidable. The property is proven by the following lemma:

Lemma 4.3 (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, we show a decidable algorithm to check whether one-step relation $\tau_1 \longrightarrow \tau_2$ holds. An intuitive algorithm is to reduce the type τ_1 by one step to obtain τ_1' (which is unique by Lemma 4.3), and compare if τ_1' and τ_2 are syntactically equal. Thus, checking if $\tau_1 \longrightarrow \tau_2$ is decidable and rules T_CastUp and T_CastDown are therefore decidable. We can conclude the decidability of type checking:

Lemma 4.4 (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 use one-step reduction, which already has a decidable checking algorithm according to Lemma 4.3. 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.

***n*-step cast operators** Because of the uniqueness of one-step reduction, we can generalize one-step cast operators to n-step. Suppose $e : \tau$ and we have sequences of reduction $\tau_1 \longrightarrow \tau_2 \longrightarrow \ldots \longrightarrow \tau_n \longrightarrow \tau$ and $\tau \longrightarrow \sigma_1 \longrightarrow \sigma_2 \longrightarrow \ldots \longrightarrow \sigma_n$. We can define n-step cast operators as follows:

$$\begin{aligned} \mathsf{cast}^n_{\uparrow}[\tau_1, \ldots, \tau_n]e &\triangleq \mathsf{cast}^{\uparrow}[\tau_1](\mathsf{cast}^{\uparrow}[\tau_2](\ldots(\mathsf{cast}^{\uparrow}[\tau_n]e)\ldots)) \\ \mathsf{cast}^n_{\downarrow} e &\triangleq \underbrace{\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow}(\ldots(\mathsf{cast}_{\downarrow}}_{n} e)\ldots)) \end{aligned}$$

By rules T_CastUp and T_CastDown, we have the following typing results:

$$\begin{aligned} \mathsf{cast}^n_{\uparrow}[\tau_1, \ldots, \tau_n]e &: \tau_1 \\ \mathsf{cast}^n_{\downarrow} e &: \sigma_n \end{aligned}$$

From Lemma 4.3, we immediately have the following corollary for n-step reduction:

Lemma 4.5 (Uniqueness of n-step reduction)**.** *The n-step reduction $\longrightarrow_n$ is unique in the sense that given e there is at most one e' such that $e \longrightarrow_n e'$.*

Proof. Immediate from Lemma 4.3, by induction on the number of reduction steps. □

Thus, $\tau_1 \longrightarrow_n \tau$ and $\tau \longrightarrow_n \sigma_n$ are unique by Lemma 4.5. The intermediate types in $\tau_1 \longrightarrow_n \tau$, i.e. $\tau_2, \ldots, \tau_n$, can be uniquely determined. Thus, we can leave them out in the $\mathsf{cast}^n_{\uparrow}$ operator. Finally, we can have n-step cast operators with the following form:

$$\begin{aligned} \mathsf{cast}^n_{\uparrow}\,[\tau_1]\, e &: \tau_1 \\ \mathsf{cast}^n_{\downarrow}\, e &: \sigma_n \end{aligned}$$

Type-safety Proof of the type-safety (or soundness) of $\lambda_\star$ 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 4.6 (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 4.7 (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 4.8 (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. $\lambda_{\star\mu}$: $\lambda_\star$ with General Recursion

In this section, we present the core calculus $\lambda_{\star\mu}$, which is the explicit core $\lambda_\star$ extended with general recursion. The general recursion is polymorphic and has a uniform representation on both term level and type level, which works as a fixpoint and recursive type respectively. We have shown that $\lambda_\star$ does not rely on normalization for decidable type checking and type-safety. Though general recursion breaks the normalizing property, it is safe for $\lambda_{\star\mu}$ to allow non-termination on the type level without breaking the decidability of type checking.

$$\begin{array}{lcll} e,\ \tau,\ \sigma & ::= & & \text{Expressions} \\ & \mid & \ldots & \\ & \mid & \mu\, x : \tau.e & \text{Polymorphic recursion} \end{array}$$

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

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

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

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

Figure 5. Grammar changes for general recursion

5.1 Grammar Changes

Figure 5 shows the changes of extending $\lambda_\star$ to $\lambda_{\star\mu}$ with general recursion. The changes are subtle, since we add only one more primitive, reduction rule and typing rule for general recursion.

For syntax, we add the polymorphic recursion operator μ to represent general recursion on both term and type level in the same form $\mu\, x : \tau.e$. For operational semantics, we add the rule S_Mu to define the unrolling operation of a recursion, which results in $e[x \mapsto \mu\, x : \tau.e]$. For typing, we add the rule T_Mu for checking the validity of a polymorphic recursive term. The rule ensures that the polymorphic recursion $\mu\, x : \tau.e$ should have the same type τ as the binder x and also the inner expression e.

5.2 Recursion as Term and Type

Term-level recursion Note that the definition of values is not changed and μ-term is not treated as a value. This is different from conventional iso-recursive types, because the μ-operator can also be regarded as a *fixpoint* on the term level. By rule S_Mu, evaluating a term $\mu\, x : \tau.e$ will substitute all x in e with the whole μ-term itself, resulting in the unrolling $e[x \mapsto \mu\, x : \tau.e]$. So the μ-term is equivalent to a recursive function that should be allowed to unroll without restriction (see example in §**??**).

Type-level recursion On the type level, $\mu\, x : \tau.e$ works as a *iso-recursive* type, a kind of recursive type that is not equal but only isomorphic to its unrolling. Normally, we need to add two more primitives fold and unfold for the iso-recursive type to map back and forth between the original and unrolled form. However, in $\lambda_{\star\mu}$ we can show that with the rule S_Mu, $\mathsf{cast}^{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ have the same functionality as fold and unfold.

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:

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

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.

5.3 Decidability of Type Checking

We give the proof of the decidability of type checking $\lambda_{\star}$ in §4.4 without requiring normalization. The cast rules are critical for decidability, which rely on checking if one type can be reduced to another in one step. When we introduce general recursion into the language, if we can make sure the newly added and original typing rules are still decidable, the decidability of type checking will still follow in $\lambda_{\star\mu}$.

The rule T_Mu for checking the well-formedness of polymorphic recursion is decidable because its premises only include decidable typing judgements. However, the rule S_Mu changes one-step reduction, which may affect the decidability of cast rules. If the uniqueness of changed reduction rules still holds, by following the same proof tactic of $\lambda_{\star}$, we can show that cast rules are still decidable in $\lambda_{\star\mu}$. Note that given a recursive term $\mu\, x : \tau.e$, by rule S_Mu, there always exists a unique term $e' = e[x \mapsto \mu\, x : \tau.e]$ such that $\mu\, x : \tau.e \longrightarrow e'$. Thus, the uniqueness of one-step reduction still holds, i.e. Lemma 4.3 holds in $\lambda_{\star\mu}$. So the decidability of type checking, namely Lemma 4.4 holds in $\lambda_{\star\mu}$.

Moreover, it is straightforward to show the type-safety of $\lambda_{\star\mu}$ by considering rules T_Mu and T_Mu during the induction of proof. Thus, Lemma 4.6, 4.7 and 4.8 also hold in $\lambda_{\star\mu}$.

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 $\lambda_{\star\mu}$, all these features can be elaborated into the core language without extending the built-in language constructs of $\lambda_{\star\mu}$. 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 $\lambda_{\star\mu}$ expressions. Finally we demonstrate the translation using a simple example.

6.1 Extended Syntax

The syntax of λC_{suf} is shown in Figure 6. Compared with $\lambda_{\star\mu}$, 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.

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

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

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

Figure 7. Syntactic sugar for records

6.2 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.

6.3 Translation Overview

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

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

It states that $\lambda_{\star\mu}$ 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 $\lambda_{\star\mu}$ 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

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

(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}

supposed to be prepended to the translation of the last expression to form a complete $\lambda_{\star\mu}$ expression, as specified by Rule (Pgm). Furthermore, each type constructor is translated as a lambda expression, with a recursive type as the body. Each data constructor is also translated as a lambda expression. Notice that we use cast operation in the lambda body to restore to the corresponding data-type.

The rest of the translation rules hold few surprises.

7. Related Work

- Henk [7] and one of its implementation [9] show the simplicity of the Pure Type System (PTS). [10] also tries to combine recursion with PTS.
- Zombie [3, 11] is a language with two fragments supporting logics with non-termination. It limits the β-reduction for congruence closure [12].
- $\Pi\Sigma$ [1] is a simple, dependently-typed core language for expressing high-level constructions[1]. UHC compiler [8] tries to use a simplified core language with coercion to encode GADTs.
- System F_C [13] has been extended with type promotion [16] and kind equality [15]. The latter one introduces a limited form of dependent types into the system[2], which mixes up types and kinds.

8. Conclusion

Conclusion and related work.

Acknowledgments

Thanks to Blah. This work is supported by Blah.

References

[1] T. Altenkirch, N. A. Danielsson, A. Löh, and N. Oury. $\Pi\Sigma$: Dependent types without the sugar. In *Functional and Logic Programming*, pages 40–55. Springer, 2010.

[2] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[3] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. *ACM SIGPLAN Notices*, 49(1):33–45, 2014.

[4] T. Coquand. *Une théorie des constructions*. PhD thesis, 1985.

[5] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, Feb. 1988. ISSN 0890-5401. . URL `http://dx.doi.org/10.1016/0890-5401(88)90005-3`.

[6] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96*, pages 284–294, 1996. ISBN 0897917693. . URL `http://dl.acm.org/citation.cfm?id=237721.237792`.

[7] S. P. Jones and E. Meijer. Henk: a typed intermediate language. 1997.

[8] A. Middelkoop, A. Dijkstra, and S. D. Swierstra. A lean specification for gadts: system f with first-class equality proofs. *Higher-Order and Symbolic Computation*, 23(2):145–166, 2010.

[9] J.-W. Roorda and J. Jeuring. Pure type systems for functional programming. 2007.

[10] P. G. Severi and F.-J. J. de Vries. Pure type systems with corecursion on streams: from finite to infinitary normalisation. In *ACM SIGPLAN Notices*, volume 47, pages 141–152. ACM, 2012.

[11] V. Sjöberg. *A Dependently Typed Language with Nontermination*. PhD thesis, University of Pennsylvania, 2015.

[12] V. Sjöberg and S. Weirich. Programming up to congruence. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 369–382, New York, NY, USA, 2015. ACM. .

[13] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.

[14] J. C. Vanderwaart, D. Dreyer, L. Petersen, K. Crary, R. Harper, and P. Cheng. *Typed compilation of recursive datatypes*, volume 38. ACM, 2003.

[15] S. Weirich, J. Hsu, and R. A. Eisenberg. Towards dependently typed haskell: System fc with kind equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP*, volume 13. Citeseer, 2013.

[16] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.

[1] But the paper didn't give any meta-theories about the langauge.

[2] Richard A. Eisenberg is going to implement kind equality [15] 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`.

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

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

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

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

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

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

$$\theta = [\overline{u := v}]$$

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

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

(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 $\lambda_{\star\mu}$

A. Full 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
	$\mid$	$\Pi\, x : \tau.\sigma$	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
Γ	$::=$		Contexts
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Gamma, x : \tau$	Variable binding
v	$::=$		Values

	$\star$	Type of types
	$\lambda x : \tau.e$	Abstraction
	$\Pi\, x : \tau_1.\tau_2$	Product
	$\mathsf{cast}^{\uparrow}\,[\tau]\,e$	Cast up by expansion

Syntactic Sugar

$T_1 \to T_2$	$\triangleq$	$\Pi\, x : T_1.T_2 \quad x \notin \mathsf{FV}(T_2)$
$(x : T_1) \to T_2$	$\triangleq$	$\Pi\, x : T_1.T_2 \quad x \in \mathsf{FV}(T_2)$
$\mathsf{cast}^{n}_{\uparrow}\,[\tau_1]\,e$	$\triangleq$	$\mathsf{cast}^{\uparrow}[\tau_1](\mathsf{cast}^{\uparrow}[\tau_2](\ldots(\mathsf{cast}^{\uparrow}\,[\tau_n]\,e)\ldots))$
$\mathsf{cast}^{n}_{\downarrow}\,e$	$\triangleq$	$\underbrace{\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow}(\ldots(\mathsf{cast}_{\downarrow}}_{n}\,e)\ldots))$

A.2 Operational Semantics

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

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

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

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

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

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

A.3 Typing

$\boxed{\vdash \Gamma}$ 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. 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 [2]):

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 (Thinning lemma)**.** *Let* Γ *and* Γ' *be legal contexts such that* $\Gamma \subseteq \Gamma'$*. If* $\Gamma \vdash e : \tau$ *then* $\Gamma' \vdash e : \tau$.

Proof. By trivial induction on the derivation of $\Gamma \vdash e : \tau$. □

Lemma B.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 $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$. 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.4 (Generation lemma)**.**

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

(2) If $\Gamma \vdash e_1\; e_2 : \sigma$*, then there exist expressions* τ_1 *and* τ_2 *such that* $\Gamma \vdash e_1 : (\Pi\, x : \tau_1.\tau_2)$*,* $\Gamma \vdash e_2 : \tau_1$ *and* $\sigma \equiv \tau_2[x \mapsto e_2]$.

(3) If $\Gamma \vdash (\lambda x : \tau_1.e) : \sigma$*, then there exist an expression* τ_2 *such that* $\sigma \equiv \Pi\, x : \tau_1.\tau_2$ *where* $\Gamma \vdash (\Pi\, x : \tau_1.\tau_2) : \star$ *and* $\Gamma, x : \tau_1 \vdash e : \tau_2$.

(4) If $\Gamma \vdash (\Pi\, x : \tau_1.\tau_2) : \sigma$*, then* $\sigma \equiv \star$*,* $\Gamma \vdash \tau_1 : \star$ *and* $\Gamma, x : \tau_1 \vdash \tau_2 : \star$*.*

(5) If $\Gamma \vdash (\mu\, x : \tau.e) : \sigma$*, then* $\Gamma \vdash \tau : \star$*,* $\sigma \equiv \tau$ *and* $\Gamma, x : \tau \vdash e : \tau$*.*

(6) If $\Gamma \vdash (\mathsf{cast}^{\uparrow}\,[\tau_1]\, e) : \sigma$*, 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. 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 and Lemma B.2, it can be shown that the statement of the lemma holds and is the only possible case. □

Lemma B.5 (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.4. □

B.2 Decidability of Type Checking

Lemma B.6 (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) $\lambda x : \tau.e$, (3) $\Pi\, x : \tau_1.\tau_2$, (4) $\mathsf{cast}^{\uparrow}\,[\tau]\, e$, 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.7 (Uniqueness of n-step reduction)**.** *The* n*-step reduction* $\longrightarrow_n$ *is unique in the sense that given* e *there is at most one* e' *such that* $e \longrightarrow_n e'$*.*

Proof. Immediate from Lemma B.6, by induction on the number of reduction steps. □

Lemma B.8 (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 T_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.4 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.6, 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 B.6, 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.9 (Multi-step reduction)**.** *The relation* $\twoheadrightarrow$ *is the transitive and reflexive closure of* $\longrightarrow$*.*

Lemma B.10 (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.4(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.4(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.3, 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 B.4(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.4(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.4(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.4(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.6 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.4(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.3, 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 B.11 (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 $e = x$**:** Impossible, because the context is empty.

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

Case $e = e_1\ e_2$**:** By Lemma B.4(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.4(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.4(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.4(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 $e' \equiv e_1[x \mapsto \mu\, x : \tau.e_1]$.

□

C. Full Specification of Source Language

C.1 Syntax

See Figure 10.

C.2 Expression Typing

See Figure 11.

C.3 Translation to the Core

See Figure 12.

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
Σ	$::=$		Contexts
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Sigma, u : T$	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

$\boxed{\vdash_{\mathsf{s}} \Sigma}$ Well-formed context

$$\frac{}{\vdash_{\mathsf{s}} \varnothing} \quad \textsc{EnvS\_Empty}$$

$$\frac{\vdash_{\mathsf{s}} \Sigma \qquad \Sigma \vdash_{\mathsf{s}} T : \star}{\vdash_{\mathsf{s}} \Sigma, x : T} \quad \textsc{EnvS\_Var}$$

$\boxed{\Sigma \vdash_{\mathsf{s}} pgm : T}$ Program context

$$\frac{\overline{\Sigma_0 \vdash_{\mathsf{s}} decl : \Sigma'} \qquad \Sigma = \Sigma_0, \overline{\Sigma'} \qquad \Sigma \vdash_{\mathsf{s}} E : T}{\Sigma_0 \vdash_{\mathsf{s}} (\overline{decl}; E) : T} \quad \textsc{TSPgm\_Pgm}$$

$\boxed{\Sigma \vdash_{\mathsf{s}} decl : \Sigma'}$ Datatype declaration

$$\frac{\Sigma \vdash_{\mathsf{s}} (\overline{u : \kappa}) \to \star : \star \qquad \overline{\Sigma, D : (\overline{u : \kappa}) \to \star, \overline{u : \kappa} \vdash_{\mathsf{s}} \overline{T} \to D\overline{u} : \star}}{\Sigma \vdash_{\mathsf{s}} (\mathsf{data}\ D\overline{u : \kappa} = \overline{K\overline{T}}) : (D : (\overline{u : \kappa}) \to \star, \overline{K : (\overline{u : \kappa}) \to \overline{T} \to D\overline{u}})} \quad \textsc{TSDecl\_Data}$$

$\boxed{\Sigma \vdash_{\mathsf{s}} p \Rightarrow E : S \to T}$ Pattern typing

$$\frac{K : (\overline{u : \kappa}) \to \overline{S} \to D\overline{u} \in \Sigma \qquad \Sigma, \overline{x : S[\overline{u \mapsto v}]} \vdash_{\mathsf{s}} E : T \qquad \Sigma \vdash_{\mathsf{s}} S[\overline{u \mapsto v}] : \star}{\Sigma \vdash_{\mathsf{s}} K\overline{x : S[\overline{u \mapsto v}]} \Rightarrow E : D\overline{v} \to T} \quad \textsc{TPat\_Alt}$$

$\boxed{\Sigma \vdash_{\mathsf{s}} E : T}$ Expression typing

$$\frac{}{\varnothing \vdash_{\mathsf{s}} \star : \star} \quad \textsc{TS\_Ax}$$

$$\frac{\vdash_{\mathsf{s}} \Sigma \qquad x : T \in \Sigma}{\Sigma \vdash_{\mathsf{s}} x : T} \quad \textsc{TS\_Var}$$

$$\frac{\Sigma \vdash_{\mathsf{s}} E_1 : (\Pi\, x : T_2.T_1) \qquad \Sigma \vdash_{\mathsf{s}} E_2 : T_2}{\Sigma \vdash_{\mathsf{s}} E_1\, E_2 : T_1[x \mapsto E_2]} \quad \textsc{TS\_App}$$

$$\frac{\Sigma, x : T_1 \vdash_{\mathsf{s}} E : T_2 \qquad \Sigma \vdash_{\mathsf{s}} (\Pi\, x : T_1.T_2) : \star}{\Sigma \vdash_{\mathsf{s}} (\lambda x : T_1.E) : (\Pi\, x : T_1.T_2)} \quad \textsc{TS\_Lam}$$

$$\frac{\Sigma \vdash_{\mathsf{s}} T_1 : \star \qquad \Sigma, x : T_1 \vdash_{\mathsf{s}} T_2 : \star}{\Sigma \vdash_{\mathsf{s}} (\Pi\, x : T_1.T_2) : \star} \quad \textsc{TS\_Pi}$$

$$\frac{\Sigma, x : T \vdash_{\mathsf{s}} E : T \qquad \Sigma \vdash_{\mathsf{s}} T : \star}{\Sigma \vdash_{\mathsf{s}} (\mu\, x : T.E) : T} \quad \textsc{TS\_Mu}$$

$$\frac{\Sigma \vdash_{\mathsf{s}} E_1 : S \qquad \overline{\Sigma \vdash_{\mathsf{s}} p \Rightarrow E_2 : S \to T} \qquad \overline{\Sigma \vdash_{\mathsf{s}} S \to T : \star}}{\Sigma \vdash_{\mathsf{s}} \mathsf{case}\ E_1\ \mathsf{of}\ \overline{p \Rightarrow E_2} : T} \quad \textsc{TS\_Case}$$

$$\frac{\Sigma \vdash_{\mathsf{s}} E_2 : T \qquad \Sigma \vdash_{\mathsf{s}} E_1[x \mapsto E_2] : S}{\Sigma \vdash_{\mathsf{s}} \mathsf{let}\ x : T = E_2\ \mathsf{in}\ E_1 : S} \quad \textsc{TS\_Let}$$

Figure 11. Typing rules of source language

$\boxed{\Sigma \vdash_{\mathsf{s}} pgm : T \rightsquigarrow e}$ Program translation

$$\frac{\overline{\Sigma_0 \vdash_{\mathsf{s}} decl : \Sigma' \rightsquigarrow e_1} \qquad \Sigma = \Sigma_0, \overline{\Sigma'} \qquad \Sigma \vdash_{\mathsf{s}} E : T \rightsquigarrow e}{\Sigma_0 \vdash_{\mathsf{s}} (\overline{decl}; E) : T \rightsquigarrow \overline{e_1} \uplus e} \text{ TRPGM\_PGM}$$

$\boxed{\Sigma \vdash_{\mathsf{s}} decl : \Sigma' \rightsquigarrow e}$ Datatype translation

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

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

$\boxed{\Sigma \vdash_{\mathsf{s}} p \Rightarrow E : S \to T \rightsquigarrow e}$ Pattern translation

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

$\boxed{\Sigma \vdash_{\mathsf{s}} E : T \rightsquigarrow e}$ Expression translation

$$\frac{}{\varnothing \vdash_{\mathsf{s}} \star : \star \rightsquigarrow \star} \text{ TR\_AX}$$

$$\frac{\vdash_{\mathsf{s}} \Sigma \qquad x : T \in \Sigma}{\Sigma \vdash_{\mathsf{s}} x : T \rightsquigarrow x} \text{ TR\_VAR}$$

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

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

$$\frac{\Sigma \vdash_{\mathsf{s}} T_1 : \star \rightsquigarrow \tau_1 \qquad \Sigma, x : T_1 \vdash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2}{\Sigma \vdash_{\mathsf{s}} (\Pi\, x : T_1.T_2) : \star \rightsquigarrow \Pi\, x : \tau_1.\tau_2} \text{ TR\_PI}$$

$$\frac{\Sigma, x : T \vdash_{\mathsf{s}} E : T \rightsquigarrow e \qquad \Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau}{\Sigma \vdash_{\mathsf{s}} (\mu\, x : T.E) : T \rightsquigarrow \mu\, x : \tau.e} \text{ TR\_MU}$$

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

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

Figure 12. Translation rules of source language