A Simple Yet Expressive Calculus for Modern Functional Languages

With technical appendix

Bi Xuan, Yang Yanpeng, and Bruno C. d. S. Oliveira

The University of Hong Kong
{xbihku, yangyp}@connect.hku.hk
bruno@cs.hku.hk

Abstract Typed core (or intermediate) languages for modern functional languages, such as Haskell or ML, are becoming more and more complex. This is a natural tendency. Programmers and language designers wish for more expressive and powerful source-language constructs. In turn this requires new, more powerful constructs in core languages. Unfortunately, the added complexity means that the meta-theory and implementation of such core languages becomes significantly harder.
This paper proposes a simple yet expressive core calculus ($\lambda_\star^\mu$), which has a fraction of the language constructs of existing core languages. The key to simplicity is the combination of two ideas. The first idea is to use a Pure Type Systems (PTS) style of syntax that unifies the various syntactic levels of the language. However, this creates an immediate challenge: with types and terms unified, the *decidability* of type checking requires type-level computation to terminate, but with general recursion it is hard to have such guarantee. The second idea, inspired by the traditional treatment of iso-recursive types, is to solve this challenge by making each type-level computation step explicit. The usefulness of $\lambda_\star^\mu$ is illustrated by a light surface language built on top of $\lambda_\star^\mu$, which supports many advanced programming language features of state-of-the-art functional languages.

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 to pick a variant of System F [14, 25] 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 [14], needs to be used. Of course the drive to add more and more advanced type-level features means that eventually the core language needs to be extended again. Indeed modern functional languages like Haskell use specially crafted core languages, such as System F_C [30], that provide support for all modern features of Haskell. Although *extensions* of System F_C [12,34] satisfy the current needs of modern Haskell, it is very likely to be extended again in the future [33]. Moreover System F_C has grown to be a relatively large and complex language, with multiple syntactic levels, and dozens of language constructs.

The more expressive type (and kind) 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 reason, implement and maintain. At the same time the core language becomes more expressive, giving us for free many useful language features. Moreover, due to the inherent expressiveness, extensions are less likely to be required. *Pure type systems* (PTS) [3] build on such observations and 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 [10]. Because the idea of using a unified syntax is so appealing several researchers have in the past considered such an option for implementing functional languages [2, 7, 22].

However having the same syntax for types and terms can also be problematic. Usually type systems based on PTS 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. When the syntax of types and terms is the same, the decidability of type checking is usually dependent on the strong normalization of the calculus. An example is the proof of decidability of type checking for the *calculus of constructions* [10] (and other normalizing PTS), which depends on strong normalization [16]. Modern dependently typed languages such as Idris [6] and Agda [20], which are also built on a unified syntax for types and terms, require strong normalization as well: all recursive programs must pass a termination checker. An unfortunate consequence of coupling decidability of type checking and strong normalization is that adding (unrestricted) general recursion to such calculi is difficult. Indeed past work on using a simple PTS-like calculi to model functional languages with unrestricted general recursion, had to give up on decidability of type-checking [2, 7].

This paper proposes $\lambda_\star^\mu$: a simple yet expressive call-by-name variant of the calculus of constructions, which has a fraction of the language constructs of existing core languages. The key challenge solved in this work is how to define a calculus comparable in simplicity to the calculus of constructions, while featuring both general recursion and decidable type checking. The main idea, inspired by the traditional treatment of *iso-recursive types* [24], is to recover decidable type-checking by making each type-level computation step explicit, i.e., each beta reduction or expansion at the type level is controlled by a *type-safe* cast. Since single computation steps are trivially terminating, decidability of type checking is possible even in the presence of non-terminating programs at the type level. At the same time term-level programs using general recursion work as in any conventional functional languages, and can even be non-terminating.

Our motivation to develop $\lambda_\star^\mu$ is to use it as a simpler alternative to existing core languages for functional programming. We focus on traditional functional languages like ML or Haskell extended with many interesting type-level features, but perhaps not the *full power* of dependent types. 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 surface language. The surface language supports *algebraic datatypes*, *higher-kinded types*, *nested datatypes* [5], *kind polymorphism* [34] and *datatype promotion* [34]. 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.

It is worth emphasizing that $\lambda_\star^\mu$ does sacrifice having an expressive form of type equality to gain the ability of doing arbitrary general recursion at the term level. Nevertheless, the core language (System F_C) of Haskell also comes with a similarly weak notion of type equality. In both System F_C and $\lambda_\star^\mu$, type equality in $\lambda_\star^\mu$ is purely syntactic (modulo alpha-conversion).

A non-goal of the current work (although a worthy avenue for future work) is to use $\lambda_\star^\mu$ as a core language for modern dependently typed languages like Agda or Idris. In contrast to $\lambda_\star^\mu$, those languages use a more powerful notion of equality. In particular $\lambda_\star^\mu$ currently lacks full-reduction and it is unable to exploit injectivity properties when comparing two types for equality. Moreover, $\lambda_\star^\mu$ (and also System F_C) lack *logical consistency*: that is ensuring the soundness of proofs written as programs. This is in contrast to dependently typed languages, where logical consistency is typically ensured. Various researchers [8, 28, 31] have been investigating how to combine logical consistency, general recursion and dependent types. However, this is usually done by having the type system carefully control the total and partial parts of computation, making those calculi significantly more complex than $\lambda_\star^\mu$ or the calculus of constructions. In $\lambda_\star^\mu$, logical consistency is traded by the simplicity of the system.

In summary, the contributions of this work are:

- **The $\lambda_\star^\mu$ calculus:** A simple core calculus for functional programming, that collapses terms, types and kinds into the same hierarchy and supports general recursion. $\lambda_\star^\mu$ is type-safe and the type system is decidable.
- **One-step casts and a generalization of iso-recursive types:** $\lambda_\star^\mu$ generalizes iso-recursive types by making all type-level computation steps explicit via *one-step casts*. In $\lambda_\star^\mu$ the combination of one-step casts and recursion subsumes iso-recursive types.
- **An expressive surface language**, built on top of $\lambda_\star^\mu$, that supports datatypes, pattern matching and various advanced language extensions of Haskell. The type safety of the type-directed translation to $\lambda_\star^\mu$ is proved.

- **A prototype implementation:** The implementation of $\lambda_\star^\mu$ is available[1].

2 Overview

This section informally introduces the main features of $\lambda_\star^\mu$. In particular, this section shows how the 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 Section 4.

2.1 The Calculus of Constructions and the Conversion Rule

The calculus of constructions (λC) [10] is a higher-order typed lambda calculus supporting dependent types (among various other features). A crucial feature of λC is the *conversion rule* :

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

The conversion rule allows one to derive $e : \tau_2$ from the derivation of $e : \tau_1$ and the beta equality of τ_1 and τ_2. This rule is important to *automatically* allow terms with beta equivalent types to be considered type-compatible. The following example illustrates the use of the conversion rule:

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

Here f is an identity function. Notice that the type of y $((\lambda x : \star.\ x)\, \mathit{Int})$ is interesting: it is a type-level identity function, applied to *Int*. Without the conversion rule, f cannot be applied to 3 for example, since the type of 3 (Int) differs from the type of y. However, note that the following beta equivalence holds:

$$(\lambda x : \star.\ x)\, \mathit{Int} =_\beta \mathit{Int}$$

Thus, the conversion rule allows the application of f to 3 by converting f to

$$\lambda y : \mathit{Int}.\ y$$

Decidability of Type Checking and Strong Normalization While the conversion rule in λC brings a lot of convenience, an unfortunate consequence is that it couples decidability of type checking with strong normalization of the calculus [16]. Therefore adding general recursion to λC becomes difficult, since strong normalization is lost. Due to the conversion rule, any non-terminating term would force the type checker to go into an infinite loop (by constantly applying the conversion rule without termination), thus rendering the type system undecidable. For example, assume a term z that has type loop, where loop stands for any diverging computation. If we type check the following application

$$(\lambda x : \mathit{Int}.\ x)\, z$$

[1] `https://github.com/bixuanzju/full-version`

under the normal typing rules of λC, the type checker would get stuck as it tries to do beta equality on two terms: Int and loop, where the latter is non-terminating.

2.2 An Alternative to the Conversion Rule: Casts

In contrast to the conversion rule 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: $\mathsf{cast}_\downarrow$ (beta reduction) and $\mathsf{cast}^\uparrow$ (beta expansion). The benefit of this approach is that decidability of type checking is no longer coupled with strong normalization of the calculus.

Beta Reduction The $\mathsf{cast}_\downarrow$ operator allows a type conversion provided that the resulting type is a *beta reduction* of the original type of the term. To explain the use of $\mathsf{cast}_\downarrow$, assume an identity function g defined by

$$g \equiv \lambda y : \mathit{Int}.\ y$$

and a term e such that

$$e : (\lambda x : \star.\ x)\,\mathit{Int}$$

In contrast to λC, we cannot directly apply g to e in $\lambda_\star^\mu$ since the type of e ($(\lambda x : \star.\ x)\,\mathit{Int}$) is not *syntactically equal* to Int. However, note that the following beta reduction holds:

$$(\lambda x : \star.\ x)\,\mathit{Int} \longrightarrow \mathit{Int}$$

Thus, $\mathsf{cast}_\downarrow$ can be used for the explicit (type-level) reduction:

$$(\mathsf{cast}_\downarrow\ e) : \mathit{Int}$$

Then the application $g\,(\mathsf{cast}_\downarrow\ e)$ type checks.

Beta Expansion The dual operation of $\mathsf{cast}_\downarrow$ is $\mathsf{cast}^\uparrow$, which allows a type conversion provided that the resulting type is a *beta expansion* of the original type of the term. To explain the use of $\mathsf{cast}^\uparrow$, let us revisit the example from Section 2.1. This time we cannot directly apply f to 3. Instead, we must use $\mathsf{cast}^\uparrow$ to expand the type of 3:

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

Thus, the application $f\ (\mathsf{cast}^\uparrow\,[(\lambda x : \star.\ x)\,\mathit{Int}]\,3)$ is well-typed. Intuitively, $\mathsf{cast}^\uparrow$ performs beta expansion, as the type of 3 is Int, and $(\lambda x : \star.\ x)\,\mathit{Int}$ is the beta expansion of Int witnessed by

$$(\lambda x : \star.\ x)\,\mathit{Int} \longrightarrow \mathit{Int}$$

Notice that for $\mathsf{cast}^\uparrow$ to work, we need to provide the resulting type as argument. This is because for the same term, there may be more than one choice for beta expansion. For example, $1 + 2$ and $2 + 1$ are both the beta expansions of 3.

One-Step The cast rules allow only *one-step* reduction or expansion. If two type-level terms require more than one step of reductions or expansions for normalization, then multiple casts must be used. Consider a variant of the example from Section 2.2. This time, assume a term e with type

$$(\lambda x : \star.\ \lambda y : \star.\ x)\, Int\, Bool$$

which is a type-level constant function. Now the following application

$$g\, (\mathsf{cast}_{\downarrow}\ e)$$

still results in an ill-typed expression, because $\mathsf{cast}_{\downarrow}\ e$ has type $(\lambda y : \star.\ Int)\ Bool$, which is not syntactically equal to Int. Thus, another $\mathsf{cast}_{\downarrow}$ is needed:

$$g\, (\mathsf{cast}_{\downarrow}\ (\mathsf{cast}_{\downarrow}\ e))$$

to further reduce $(\lambda y : \star.\ Int)\ Bool$ to Int, allowing the program to type check.

Decidability without Strong Normalization With explicit type conversion rules the decidability of type checking no longer depends on the strong normalization property. Thus the type system remains decidable even in the presence of non-termination at type level. Consider the same example in Section 2.1. This time the type checker will not get stuck when type checking the following application:

$$(\lambda x : Int.\ x)\, z$$

This is because in $\lambda^{\mu}_{\star}$, the type checker only performs syntactic comparison between Int and loop, instead of beta equality. Thus it rejects the above application as ill-typed. Indeed it is impossible to type check such application 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., $(\lambda x : Int.\ x)(\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow} \ldots z)))$. But it is impossible to write such program in practice.

In summary, $\lambda^{\mu}_{\star}$ achieves decidability of type checking by explicitly controlling type-level computation using casts. Since each cast performs only one step of computation at a time, type-level computation performed by each cast is guaranteed to terminate.

2.3 Recursive Terms and Types

$\lambda^{\mu}_{\star}$ supports general recursion, and allows writing 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, $\lambda^{\mu}_{\star}$ *unifies* both recursion and recursive types by the same μ primitive.

Recursive Terms The primitive $\mu\, x : \tau.\ e$ can be used to define recursive functions. For example, the factorial function is written in $\lambda^{\mu}_{\star}$ as:

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

Here we treat the μ operator as a *fixpoint*, which evaluates to its recursive unfolding:

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

Term-level recursion in $\lambda_\star^\mu$ works as in any standard functional language. The application *fact* 3, for example, produces 6 as expected.

Recursive Types The same μ primitive is used at the type level to represent iso-recursive types [11]. In the *iso-recursive* approach a recursive type and its unfolding are different, but isomorphic. The isomorphism is witnessed by two operations, typically called fold and unfold. In $\lambda_\star^\mu$, such isomorphism is witnessed by $\mathsf{cast}^\uparrow$ and $\mathsf{cast}_\downarrow$. In fact, $\mathsf{cast}^\uparrow$ and $\mathsf{cast}_\downarrow$ *generalize* fold and unfold: they can convert any types, not just recursive types.

To demonstrate the use of the cast rules with recursive types, we show the formation of the "hungry" type [24] $H = \mu\, x : \star.\ Int \to x$. A term z of type H accepts any number of integers and returns a new function that is hungry for more, as illustrated below:

$$\begin{aligned}(\mathsf{cast}_\downarrow\, z)\, 3 &: H \\ \mathsf{cast}_\downarrow\, ((\mathsf{cast}_\downarrow\, z)\, 3)\, 3 &: H \\ \mathsf{cast}_\downarrow (\ldots (\mathsf{cast}_\downarrow\, z)\, 3 \ldots)\, 3 &: H\end{aligned}$$

3 Fun: A Surface Language on Top of $\lambda_\star^\mu$

The main goal of $\lambda_\star^\mu$ is to serve as an expressive core language for functional languages like Haskell or ML. This section shows a number of programs written in the surface language **Fun**, which is built on top of $\lambda_\star^\mu$. We illustrate the expressiveness of $\lambda_\star^\mu$ by encoding functional programs that require some of the latest features of Haskell, or are non-trivial to encode in dependently typed language like Coq or Agda. All examples shown in this section are runnable in our prototype interpreter. The formalization of the surface language is presented in Section 5.

Datatypes Conventional datatypes like natural numbers or polymorphic lists can be easily defined in **Fun**. For instance, below is the definition of polymorphic lists:

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

The use of the above datatype is illustrated by the *length* function:

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

The *length* function is recursive. **Fun** supports a standard **letrec** construct that facilitates defining recursive functions. The return type of *length* is *int*, the built-in integer type. Note that due to explicit typing, the program requires quite a few type annotations and type parameters. However, apart from the extra typing, the program is similar to the code that would be written in a language like Haskell or ML.

HOAS Higher-order abstract syntax [23] is a representation of abstract syntax where the function space of the meta-language is used to encode the binders of the object language. We show an example of encoding a simple lambda calculus:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Exp} = \mathit{Num}\ (n : \mathit{int})\\
&\quad\mid \mathit{Lam}\ (f : \mathit{Exp} \to \mathit{Exp})\\
&\quad\mid \mathit{App}\ (a : \mathit{Exp})\ (b : \mathit{Exp});
\end{aligned}
$$

Note that in the lambda constructor (Lam), the recursive occurrence of Exp appears in a negative position (i.e., in the left side of a function arrow). Systems like Coq and Agda would reject such programs since it is well-known that such datatypes can lead to logical inconsistencies. Moreover, such logical inconsistencies can be exploited to write non-terminating computations, and make type checking undecidable. However **Fun** is able to express HOAS in a straightforward way, while preserving decidable type checking.

Using *Exp* we can write an evaluator for the lambda calculus. As noted by Fegaras and Sheard [13], the evaluation function needs an extra function (*reify*) to invert the result of evaluation. The code for the evaluator is shown next (we omit most of the unsurprising cases):

$$
\begin{aligned}
&\textbf{data}\ \mathit{Value} = \mathit{VI}\ (n : \mathit{int}) \mid \mathit{VF}\ (f : \mathit{Value} \to \mathit{Value});\\
&\textbf{rcrd}\ \mathit{Eval} = \mathit{Ev}\ \{\mathit{eval}' : \mathit{Exp} \to \mathit{Value}, \mathit{reify}' : \mathit{Value} \to \mathit{Exp}\};\\
&\textbf{letrec}\ \mathit{ev} : \mathit{Eval} =\\
&\quad \mathit{Ev}\ (\lambda e : \mathit{Exp}.\, \textbf{case}\ e\ \textbf{of}\\
&\qquad\quad \mid \ldots\\
&\qquad\quad \mid \mathit{Lam}\ (\mathit{fun} : \mathit{Exp} \to \mathit{Exp}) \Rightarrow\\
&\qquad\qquad \mathit{VF}\ (\lambda e' : \mathit{Value}.\, \mathit{eval}'\ \mathit{ev}\ (\mathit{fun}\ (\mathit{reify}'\ \mathit{ev}\ e')))\\
&\qquad\quad (\lambda v : \mathit{Value}.\, \textbf{case}\ v\ \textbf{of}\\
&\qquad\quad \mid \ldots\\
&\qquad\quad \mid \mathit{VF}\ (\mathit{fun} : \mathit{Value} \to \mathit{Value}) \Rightarrow\\
&\qquad\qquad \mathit{Lam}\ (\lambda e' : \mathit{Exp}.\, \mathit{reify}'\ \mathit{ev}\ (\mathit{fun}\ (\mathit{eval}'\ \mathit{ev}\ e')))\\
&\textbf{in let}\ \mathit{eval} : \mathit{Exp} \to \mathit{Value} = \mathit{eval}'\ \mathit{ev}
\end{aligned}
$$

The definition of the evaluator is mostly straightforward. Here we create a record *Eval* (by using **rcrd** keyword), inside which are two mutually recursive functions eval' and reify'. The former one is conventional, dealing with each possible shape of an expression. The tricky part lies in the evaluation of a lambda abstraction, where we need a second function, called reify', of type $\mathit{Value} \to \mathit{Exp}$ that lifts a values into terms. Thanks to the flexibility of the μ primitive, mutual recursion can be encoded by using records.

Evaluation of a lambda expression proceeds as follows:

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

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 use core languages to account for kind expressions. The existing core language of Haskell, System F_C, is an extension of System F_ω [14], which natively supports higher-kinded types. Given that $\lambda_\star^\mu$ subsumes System F_ω, we can easily construct higher-kinded types. We show with an example of encoding the *functor* "type-class" as a record:

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

Here we use a record to represent a functor, whose only field is a function called *fmap*. The functor "instance" of the *Maybe* datatype is:

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

After the translation process, the *Functor* record is desugared into a datatype with only one data constructor (*Func*) that has type:

$$(f : \star \to \star) \to (a : \star) \to (b : \star) \to (a \to b) \to f\ a \to f\ b$$

Since *Maybe* has kind $\star \to \star$, it is legal to apply *Func* to *Maybe*. The definition of *fmap* is straightforward.

Nested Datatypes A nested datatype [5], also known as a *non-regular* datatype, is a parameterized datatype whose definition contains different instances of the type parameters. Functions over nested datatypes usually involve polymorphic recursion. We show that **Fun** is capable of defining nested datatypes and 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:

$$\begin{array}{l}\textbf{data}\ \mathit{PairT}\ (a : \star) = P\ (x : a)\ (x : a);\\ \textbf{data}\ \mathit{Pow}\ (a : \star) = \mathit{Zero}\ (n : a) \mid \mathit{Succ}\ (t : \mathit{Pow}\ (\mathit{PairT}\ a));\end{array}$$

Notice that the recursive occurrence of *Pow* does not hold a, but $\mathit{PairT}\ a$. This means every time we use a *Succ* constructor, the size of the pairs doubles. It is instructive to look at the encoding of *Pow* in $\lambda_\star^\mu$:

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

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

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

Kind Polymorphism Previous versions of Haskell, based on System F_ω, had a simple kind system with a few kinds ($\star$, $\star \to \star$ and so on). Unfortunately, this was insufficient for kind polymorphism. Thus, recent versions of Haskell were extended to support kind polymorphism, which required extending the core language as well. Indeed, System $F_C^\uparrow$ [34] was proposed to support, among other things, kind polymorphism. However, System $F_C^\uparrow$ separates expressions into terms, types and kinds, which complicates both the implementation and future extensions. In contrast, without additional extensions, **Fun** natively supports kind polymorphism. Here is an example, taken from [34], of a datatype that benefits from kind polymorphism: a higher-kinded fixpoint combinator:

$$\textbf{data}\ \mathit{Mu}\ (k : \star)\ (f : (k \to \star) \to k \to \star)\ (a : k) = \mathit{Roll}\ (g : f\ (\mathit{Mu}\ k\ f)\ a);$$

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

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

Datatype Promotion Recent versions of Haskell introduced datatype promotion [34], allowing ordinary datatypes promoted as kinds, and data constructors as types. With the power of dependent types, datatype promotion is made trivial in **Fun**.

As a last example, we show a representation of a labeled binary tree, where each node is labeled with its depth in the tree. Below is the definition:

$$\begin{array}{l}\textbf{data}\ \mathit{PTree}\ (n : \mathit{Nat}) = \mathit{Empty}\\ \quad \mid \mathit{Fork}\ (z : \mathit{int})\ (x : \mathit{PTree}\ (S\ n))\ (y : \mathit{PTree}\ (S\ n));\end{array}$$

Notice how the datatype *Nat* is "promoted" to be used in the kind level. Next we can construct a binary tree that keeps track of its depth statically:

Fork Z 1 (*Empty* (S Z)) (*Empty* (S Z))

If we accidentally write the wrong depth, for example:

Fork Z 1 (*Empty* (S Z)) (*Empty* Z)

The above will fail to pass type checking.

4 Dependent Types with Casts and General Recursion

In this section, we present the $\lambda_\star^\mu$ calculus. This calculus is very close to the calculus of constructions, except for three key differences: 1) the absence of the $\square$ constant (due to use of the "type-in-type" axiom); 2) the existence of two cast operators; 3) general recursion on both term level and type level. Unlike λC the proof of decidability of type checking for $\lambda_\star^\mu$ does not require the strong normalization of the calculus. Thus, the addition of general recursion does not break decidable type checking. In the rest of this section, we demonstrate the syntax, operational semantics, typing rules and metatheory of $\lambda_\star^\mu$. Full proofs of metatheory can be found in the full version of this paper[2].

4.1 Syntax

Figure 1 shows the syntax of $\lambda_\star^\mu$, including expressions, contexts and values. $\lambda_\star^\mu$ uses a unified representation for different syntactic levels by following the *pure type system* (PTS) representation of λC [3]. Therefore, there is no syntactic distinction between terms, types or kinds. This design brings 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.

Type of Types In λC, there are two distinct sorts $\star$ and $\square$ representing the type of *types* and *sorts* respectively, and an axiom $\star : \square$ specifying the relation between the two sorts [3]. In $\lambda_\star^\mu$, we further merge types and kinds together by including only a single sort $\star$ and an impredicative axiom $\star : \star$.

Explicit Type Conversion We introduce two new primitives $\mathsf{cast}^\uparrow$ and $\mathsf{cast}_\downarrow$ (pronounced as "cast up" and "cast down") to replace the implicit conversion rule of λC with *one-step* explicit type conversions. The type-conversions perform two directions of conversion: $\mathsf{cast}_\downarrow$ is for the *beta reduction* of types, and $\mathsf{cast}^\uparrow$ is for the *beta expansion*. The $\mathsf{cast}^\uparrow$ construct takes a type parameter τ as the result type of one-step beta expansion for disambiguation (see also Section 2.2). The $\mathsf{cast}_\downarrow$ construct does not need a type parameter, because the result type of one-step beta reduction is uniquely determined, as we shall see in Section 4.5.

[2] `https://github.com/bixuanzju/full-version`

$e,\ \tau,\ \sigma,\ \rho$	$::=$		Expressions
	$\mid$	x	Variable
	$\mid$	$\star$	Type of Types
	$\mid$	$e_1\ e_2$	Application
	$\mid$	$\lambda x : \tau.\ e$	Abstraction
	$\mid$	$\Pi\, x : \tau.\ \sigma$	Dependent Product
	$\mid$	$\mathsf{cast}^{\uparrow}\,[\tau]\ e$	Cast Up
	$\mid$	$\mathsf{cast}_{\downarrow}\ e$	Cast Down
	$\mid$	$\mu\, x : \tau.\ e$	Polymorphic Recursion
Γ	$::=$		Contexts
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Gamma, x : \tau$	Variable Binding
v	$::=$		Values
	$\mid$	$\star$	Type of Types
	$\mid$	$\lambda x : \tau.\ e$	Abstraction
	$\mid$	$\Pi\, x : \tau_1.\ \tau_2$	Dependent Product
	$\mid$	$\mathsf{cast}^{\uparrow}\,[\tau]\ e$	Cast Up

Syntactic Sugar

$\tau_1 \to \tau_2$	$\triangleq \Pi\, x : \tau_1.\ \tau_2,\ \text{where } x \notin \mathsf{FV}(\tau_2)$
$(x : \tau_1) \to \tau_2$	$\triangleq \Pi\, x : \tau_1.\ \tau_2$
$\mathbf{let}\, x : \tau = e_2\ \mathbf{in}\ e_1$	$\triangleq e_1[x \mapsto e_2]$
$\mathbf{letrec}\, x : \tau = e_2\ \mathbf{in}\ e_1$	$\triangleq \mathbf{let}\, x : \tau = \mu\, x : \tau.\ e_2\ \mathbf{in}\ e_1$
$\mathsf{cast}^{n}_{\uparrow}\,[\tau_1]\ e$	$\triangleq \mathsf{cast}^{\uparrow}[\tau_1](\mathsf{cast}^{\uparrow}[\tau_2](\ldots(\mathsf{cast}^{\uparrow}\,[\tau_n]\ e)\ldots))$
$\mathsf{cast}^{n}_{\downarrow}\ e$	$\triangleq \underbrace{\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow}(\ldots(\mathsf{cast}_{\downarrow}}_{n}\ e)\ldots))$

Figure 1. Syntax of $\lambda^{\mu}_{\star}$

General Recursion General recursion allows a large number of programs that can be expressed in programming languages such as Haskell to be expressed in $\lambda^{\mu}_{\star}$ as well. We add one primitive μ to represent general recursion. It has a uniform representation on both term level and type level: the same construct works both as a term-level fixpoint and a recursive type. The recursive expression $\mu\, x : \tau.\ e$ is *polymorphic*, in the sense that τ is not restricted to $\star$ but can be any type, such as a function type $\mathit{Int} \to \mathit{Int}$ or a kind $\star \to \star$.

Syntactic Sugar Figure 1 also shows the syntactic sugar used in $\lambda^{\mu}_{\star}$. By convention, we use $\tau_1 \to \tau_2$ to represent $\Pi\, x : \tau_1.\ \tau_2$ if x does not occur free in τ_2. We also interchangeably use the dependent function type $(x : \tau_1) \to \tau_2$ to denote $\Pi\, x : \tau_1.\ \tau_2$. We use $\mathbf{let}\, x : \tau = e_2\ \mathbf{in}\ e_1$ to locally bind a variable x to an expression e_2 in e_1, and its variant **letrec** for recursive functions.

For the brevity of translation rules in Section 5, we use $\mathsf{cast}^{n}_{\uparrow}$ and $\mathsf{cast}^{n}_{\downarrow}$ to denote n consecutive cast operators. $\mathsf{cast}^{n}_{\uparrow}$ is simplified to only take one type parameter, the last type τ_1 of the n cast operations. The original $\mathsf{cast}^{n}_{\uparrow}$ includes intermediate results $\tau_2, \ldots, \tau_n$ of type conversion:

$$\mathsf{cast}^{n}_{\uparrow}[\tau_1, \ldots, \tau_n]e \triangleq\ \mathsf{cast}^{\uparrow}[\tau_1](\mathsf{cast}^{\uparrow}[\tau_2](\ldots(\mathsf{cast}^{\uparrow}[\tau_n]e)\ldots))$$

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

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

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

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

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

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

Figure 2. Operational semantics of $\lambda^{\mu}_{\star}$

Due to the decidability of one-step reduction (shown in Section 4.5), the intermediate types can be uniquely determined, thus can be left out from the $\mathsf{cast}^{n}_{\uparrow}$ operator.

4.2 Operational Semantics

Figure 2 shows the *call-by-name* operational semantics, defined by one-step reduction. Three base cases include S_Beta for beta reduction, S_CastDownUp for cast canceling and S_Mu for recursion unrolling. Two inductive cases, 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* because they are not allowed to do the reduction inside a λ-term or $\mathsf{cast}^{\uparrow}$-term — both of them are defined as values (see Figure 1).

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. So we can define the *multi-step reduction*:

Definition 1 (Multi-step reduction). *The relation $\twoheadrightarrow$ is the transitive and reflexive closure of the one-step reduction $\longrightarrow$.*

4.3 Typing

Figure 3 gives the *syntax-directed* typing rules of $\lambda^{\mu}_{\star}$, 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. 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 respectively. Rule T_Pi checks the type well-formedness of the dependent function. Rule T_Mu checks the validity of a recursive term. It ensures that the recursion $\mu\, x : \tau .\ e$ should have the same type τ as the binder x and also the inner expression e.

The Cast Rules We focus on rules T_CastUp and T_CastDown that define the semantics of cast operators and replace the conversion rule of λC. The relation between the original and converted type is defined by one-step reduction (see Figure 2).

For example, given a judgement $\Gamma \vdash e : \tau_2$ and relation $\tau_1 \longrightarrow \tau_2 \longrightarrow \tau_3$, $\mathsf{cast}^{\uparrow}\,[\tau_1]\; e$ 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. We can formally give the typing derivations of the examples in Section 2.2:

$$\frac{\Gamma \vdash e : (\lambda x : \star .\ x)\ \mathit{Int} \qquad \Gamma \vdash \mathit{Int} : \star \qquad (\lambda x : \star .\ x)\ \mathit{Int} \longrightarrow \mathit{Int}}{\Gamma \vdash (\mathsf{cast}_{\downarrow}\; e) : \mathit{Int}}$$

$$\frac{\Gamma \vdash 3 : \mathit{Int} \qquad \Gamma \vdash (\lambda x : \star .\ x)\ \mathit{Int} : \star \qquad (\lambda x : \star .\ x)\ \mathit{Int} \longrightarrow \mathit{Int}}{\Gamma \vdash (\mathsf{cast}^{\uparrow}\,[(\lambda x : \star .\ x)\ \mathit{Int}]\; 3) : (\lambda x : \star .\ x)\ \mathit{Int}}$$

Importantly, in $\lambda^{\mu}_{\star}$ term-level and type-level computation are treated differently. Term-level computation is dealt in the usual way, by using multi-step reduction until a value is finally obtained. Type-level computation, on the other hand, is controlled by the program: each step of the computation is induced by a cast. If a type-level program requires n steps of computation to reach normal form, then it will require n casts to compute a type-level value.

Pros and Cons of Type in Type The "type-in-type" axiom is well-known to give rise to logical inconsistency [14]. However, since our goal is to investigate core languages for languages that are logically inconsistent anyway (due to general recursion), we do not view "type-in-type" as a problematic rule. On the other hand the rule T_Ax brings additional expressiveness and benefits: for example *kind polymorphism* is supported in $\lambda^{\mu}_{\star}$. A term that takes a kind parameter like $\lambda x : \Box .\ x \to x$ cannot be typed in λC, since $\Box$ is the highest sort that does not have a type. In contrast, $\lambda^{\mu}_{\star}$ does not have such limitation. Because of the $\star : \star$ axiom, the term $\lambda x : \star .\ x \to x$ has a legal type $\Pi\, x : \star .\ \star$ in $\lambda^{\mu}_{\star}$. It can be applied to a kind such as $\star \to \star$ to obtain $(\star \to \star) \to (\star \to \star)$.

Syntactic Equality Finally, the definition of type equality in $\lambda^{\mu}_{\star}$ differs from λC. Without λC's conversion rule, the type of a term cannot be converted freely against beta equality, unless using cast operators. Thus, types of expressions are equal only if they are syntactically equal (up to alpha renaming).

4.4 The Two Faces of Recursion

Term-level Recursion In $\lambda^{\mu}_{\star}$, the μ-operator works as a *fixpoint* on the term level. By rule S_Mu, evaluating a term $\mu\, x : \tau .\ e$ will substitute all x's in e with the whole μ-term itself, resulting in the unrolling $e[x \mapsto \mu\, x : \tau .\ e]$. The μ-term

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

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

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

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

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

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

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

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

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

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

Figure 3. Typing rules of $\lambda_\star^\mu$

is equivalent to a recursive function that should be allowed to unroll without restriction. Therefore, the definition of values is not changed in $\lambda_\star^\mu$ and a μ-term is not treated as a value.

Recall the factorial function example in Section 2.3. By rule T_Mu, the type of *fact* is $Int \to Int$. Thus we can apply *fact* to an integer. In this example, we assume evaluating the **if-then-else** construct and arithmetic expressions follows the one-step reduction. Together with standard reduction rules S_Mu and S_App, we can evaluate the term *fact* 3 as follows:

$$\begin{array}{l} \quad\ \ \mathit{fact}\ 3 \\ \longrightarrow (\lambda x : \mathit{Int}.\ \ \mathbf{if}\ \ x == 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}\ (x-1))\ 3 \\ \longrightarrow \mathbf{if}\ 3 == 0\ \mathbf{then}\ 1\ \mathbf{else}\ 3 \times \mathit{fact}\ (3-1) \\ \longrightarrow 3 \times \mathit{fact}\ (3-1) \\ \longrightarrow \ldots \longrightarrow\ 6. \end{array}$$

Note that we never check if a μ-term can terminate or not, which is an undecidable problem for general recursive terms. The factorial function example above can stop, while there exist some terms that will loop forever. However, term-level non-termination is only a runtime concern and does not block the type checker. In Section 4.5 we show type checking $\lambda_\star^\mu$ is still decidable in the presence of general recursion.

Type-level Recursion On the type level, $\mu\, x : \tau.\ e$ works as a *iso-recursive* type [11], 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. Assume there exist expressions e_1 and e_2 such that

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

We have the following typing results

$$\begin{aligned} &\mathsf{unfold}\ e_1 && : \sigma[x \mapsto \mu\, x : \tau.\ \sigma] \\ &\mathsf{fold}\ [\mu\, x : \tau.\ \sigma]\ e_2 && : \mu\, x : \tau.\ \sigma \end{aligned}$$

which are derived from the standard rules for recursive types [24]

$$\frac{\Gamma \vdash e_1 : (\mu\, x : \tau.\ \sigma) \qquad \Gamma \vdash \sigma[x \mapsto \mu\, x : \tau.\ \sigma] : \star}{\Gamma \vdash \mathsf{unfold}\ e_1 : (\sigma[x \mapsto \mu\, x : \tau.\ \sigma])}$$

$$\frac{\Gamma \vdash e_2 : \sigma[x \mapsto \mu\, x : \tau.\ \sigma] \qquad \Gamma \vdash (\mu\, x : \tau.\ \sigma) : \star}{\Gamma \vdash \mathsf{fold}\ [\mu\, x : \tau.\ \sigma]\ e_2 : (\mu\, x : \tau.\ \sigma)}$$

Thus, we have the following relation between types of e_1 and e_2 witnessed by fold and unfold:

$$\mu\, x : \tau.\ \sigma \underset{\mathsf{fold}\ [\mu\, x : \tau.\ \sigma]}{\overset{\mathsf{unfold}}{\rightleftharpoons}} \sigma[x \mapsto \mu\, x : \tau.\ \sigma]$$

However, in $\lambda_\star^\mu$ we do not need to introduce fold and unfold operators, because with the rule S_Mu, $\mathsf{cast}^\uparrow$ and $\mathsf{cast}_\downarrow$ *generalize* fold and unfold. Consider the same expressions e_1 and e_2 above. The type of e_2 is the unrolling of e_1's type, which follows the one-step reduction relation by rule S_Mu:

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

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

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

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

$$\mu\, x : \tau.\ \sigma \underset{\mathsf{cast}^\uparrow\ [\mu\, x : \tau.\ \sigma]}{\overset{\mathsf{cast}_\downarrow}{\rightleftharpoons}} \sigma[x \mapsto \mu\, x : \tau.\ \sigma]$$

Casts and Recursive Types Figure 1 shows that $\mathsf{cast}^{\uparrow}$ is a *value* that cannot be further reduced during the evaluation. It follows the convention of fold in iso-recursive types [32]. But too many $\mathsf{cast}^{\uparrow}$ constructs left for code generation will increase the size of the program and cause runtime overhead. Actually, $\mathsf{cast}^{\uparrow}$ constructs can be safely erased after type checking: they are *computationally irrelevant* and do not actually transform a term other than changing its type.

An important remark is that casts are necessary, not only for controlling the unrolling of recursive types, but also for type conversion of other constructs. This is because the "type-in-type" axiom [7] makes it possible to encode fixpoints even without a fixpoint primitive, i.e., the μ-operator. Thus if no casts would be performed on terms without recursive types, it would still be possible to build a term with a non-terminating type and make type-checking non-terminating.

4.5 Metatheory

We now discuss the metatheory of $\lambda_{\star}^{\mu}$. We focus on two properties: the decidability of type checking and the type safety of the language. First, we show that type checking $\lambda_{\star}^{\mu}$ is decidable without requiring strong normalization. Second, the language is type-safe, proven by subject reduction and progress theorems.

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, including the rule T_Mu for recursion, which have only typing judgements in premises, are already decidable by the 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 τ_2 follow 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 would not be decidable. The decidability of one-step reduction is given by the following lemma:

Lemma 1 (Decidability of One-step Reduction). *The one-step reduction $\longrightarrow$ is called decidable if given e there is a unique e' such that $e \longrightarrow e'$ or there is no such e'.*

Proof. By induction on the structure of e. □

Note that the presence of recursion does not affect this lemma: 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 $\mu\, x : \tau.\ e \longrightarrow e'$. With this result, we show a decidable algorithm to check whether the 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 1), 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:

Theorem 1 (Decidability of Type Checking $\lambda_\star^\mu$). *There is an 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. $\square$

We emphasize that when proving the decidability of type checking, we do not rely on strong normalization. Intuitively, explicit type conversion rules use one-step reduction, which already has a decidable checking algorithm according to Lemma 1. We do not need to further require the normalization of terms. This is different from the proof for λC which requires the language to be strongly normalizing [16]. In λC the conversion rule needs to examine the beta equivalence of terms, which is decidable only if every term has a normal form.

Type Safety The proof of the type safety of $\lambda_\star^\mu$ is fairly standard by subject reduction and progress theorems. The subject reduction proof relies on the substitution lemma. We give the proof sketch of the related lemma and theorems as follows:

Lemma 2 (Substitution). *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. (*Sketch*) By induction on the derivation of $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$. We only treat cases T_Mu, T_CastUp and T_CastDown since other cases can be easily followed by the proof for PTS in [3]. $\square$

Theorem 2 (Subject Reduction of $\lambda_\star^\mu$). *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 theorem follows 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 $\longrightarrow$. $\square$

Theorem 3 (Progress of $\lambda_\star^\mu$). *If $\varnothing \vdash e : \sigma$ then either e is a value v or there exists e' such that $e \longrightarrow e'$.*

Proof. By induction on the derivation of $\varnothing \vdash e : \sigma$. $\square$

5 Formalization of the Surface language

In this section, we formally present the surface language **Fun**, built on top of $\lambda_\star^\mu$ with features that are convenient for functional programming. Thanks 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 the surface language, followed by the typing rules, then we show the formal translation rules that translates a surface language expression to an expression in $\lambda_\star^\mu$. Finally we prove the type safety of the translation.

5.1 Syntax

The full syntax of **Fun** is defined in Figure 4. Compared with $\lambda_\star^\mu$, **Fun** has a new syntax category: a program, consisting of a list of datatype declarations, followed by an expression. For the purpose of presentation, we sometimes adopt the following syntactic convention:

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

Algebraic Datatypes An *algebraic datatype* D is introduced as a top-level **data** declaration with its *data constructors.* The type of a data constructor K takes the form:

$$K : (\,\overline{u : \kappa}^n\,) \to (\,\overline{x : T}\,) \to D\,\overline{u}^n$$

The first n quantified type variables $\overline{u}$ appear in the same order in the result type $D\,\overline{u}$. Note that the use of the dependent product in the data constructor arguments (i.e., $(\,\overline{x : T}\,)$) 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 (i.e., no nested patterns), and bind variables. We also introduce a Haskell-like record syntax, which are desugared to datatypes with accompanying selector functions.

No Casts on The Surface A noticeable difference from $\lambda_\star^\mu$ is that, in **Fun**, we do not allow cast operations to appear in surface programs. However, the surface language takes good advantage of the benefits of the core language. The encoding of datatypes and case analysis uses casts and type-level computation steps in a fundamental way: we need to use casts to simulate fold/unfold, and we also need small type-level computational steps to encode parametrised datatypes. Casts are mostly intended to be generated by the compiler for **Fun**, not by the programmers. An unfortunate consequence is that, this, for now, makes the surface language less expressive (e.g., no explicit type-level computation) than $\lambda_\star^\mu$. One avenue for future work is to explore the addition of good, surface level, mechanisms for doing type-level computation built on top of casts.

5.2 Typing Rules

Figure 5 defines the type system of the surface language. The gray parts can be ignored for the moment. Several new typing judgments are introduced in the type system. The use of different subscripts in the judgments is to be distinct from the ones used in $\lambda_\star^\mu$. Most rules are standard for systems based on λC, including the rules for the well-formedness of contexts (TRenv_Empty, TRenv_Var), inferring the types of variables (TR_Var), and dependent application (TR_App).

Rule TRpgm_Pgm type checks a whole program. It first type-checks the declarations, which in return gives a new typing environment. Combined with the original environment, it then continues to type check the final expression and

$$
\begin{array}{lcll}
\mathit{pgm} & ::= & & \text{Program} \\
 & \mid & \overline{\mathit{decl}}\,; E & \text{Declarations} \\[4pt]
\mathit{decl} & ::= & & \text{Declarations} \\
 & \mid & \mathbf{data}\, D\, \overline{u:\kappa} \,=\, \overline{K\, \overline{x:T}} & \text{Datatype} \\[4pt]
u,\ \upsilon & ::= & & \text{Atoms} \\
 & \mid & x & \text{Variable} \\
 & \mid & K & \text{Data Constructor} \\[4pt]
p & ::= & & \text{Patterns} \\
 & \mid & K\, \overline{x:T} & \text{Pattern} \\[4pt]
E,\ T,\ S,\ \kappa & ::= & & \text{Expressions} \\
 & \mid & u & \text{Atom} \\
 & \mid & \star & \text{Type of Types} \\
 & \mid & E_1\, E_2 & \text{Application} \\
 & \mid & \lambda x : T.\ E & \text{Abstraction} \\
 & \mid & \Pi\, x : T.\ S & \text{Dependent Product} \\
 & \mid & \mu\, x : T.\ E & \text{Recursion} \\
 & \mid & \mathbf{case}\, E_1\, \mathbf{of}\ \overline{p \Rightarrow E_2} & \text{Case Analysis} \\[4pt]
\Sigma & ::= & & \text{Contexts} \\
 & \mid & \varnothing & \text{Empty} \\
 & \mid & \Sigma, u : T & \text{Atom Binding}
\end{array}
$$

Syntactic Sugar

$$
\begin{array}{ll}
\mathbf{rcrd}\, R\, \overline{u:\kappa}^n \,=\, K\{\, \overline{N:T}\,\} \triangleq & \mathbf{data}\, R\, \overline{u:\kappa}^n \,=\, K\, \overline{x:T}\,; \\
 & \mathbf{let}\ N_i : (\, \overline{u:\kappa}^n\,) \to R\, \overline{u}^n \to T_i = \\
 & \quad \lambda\, \overline{u:\kappa}^n .\ \lambda y : R\, \overline{u}^n . \\
 & \quad \mathbf{case}\, y\, \mathbf{of}\ K\, \overline{x:T} \Rightarrow x_i\ \mathbf{in}
\end{array}
$$

Figure 4. Syntax of **Fun**

return the result type. Rule TRPGM_DATA is used to type check datatype declarations. It first ensures the well-formedness of the type of the type constructor (of sort $\star$). Then it ensures that the types of data constructors are valid. Note that since our system adopts the $\star : \star$ axiom, this means we can express kind polymorphism in datatypes.

Rules TR_CASE and TRPAT_ALT handle the type checking of case expressions. The conclusion of TS_CASE binds the scrutinee E_1 and alternatives $\overline{p \Rightarrow E_2}$ to the right types. The first premise of TS_CASE binds the actual type constructor arguments to $\overline{\upsilon}$. The second premise checks whether the types of the alternatives, instantiated to the actual type constructor arguments $\overline{\upsilon}$ (via TRPAT_ALT), are equal. Finally the third premise checks the well-formedness of the result type.

5.3 Translation Overview

We use a type-directed translation. The basic translation rules have the form:

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

$\boxed{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma}$ Context well-formedness

$$\frac{}{\vdash_{\mathsf{wf}} \varnothing \rightsquigarrow \varnothing}\ \textsc{TRenv\_Empty}$$

$$\frac{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma \qquad \Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau}{\vdash_{\mathsf{wf}} \Sigma, x : T \rightsquigarrow \Gamma, x : \tau}\ \textsc{TRenv\_Var}$$

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

$$\frac{\overline{\Sigma_0 \vdash_{\mathsf{d}} decl : \Sigma' \rightsquigarrow e_1} \qquad \Sigma = \Sigma_0, \overline{\Sigma'} \qquad \Sigma \vdash_{\mathsf{s}} E : T \rightsquigarrow e}{\Sigma_0 \vdash_{\mathsf{pg}} (\, \overline{decl}\,; E) : T \rightsquigarrow \overline{e_1} \uplus e}\ \textsc{TRpgm\_Pgm}$$

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

$$\frac{\dfrac{\Sigma \vdash_{\mathsf{s}} (\,\overline{u : \kappa}^n\,) \to \star : \star \rightsquigarrow (\,\overline{u : \rho}^n\,) \to \star}{\overline{\Sigma, D : (\,\overline{u : \kappa}^n\,) \to \star,\ \overline{u : \kappa}^n \vdash_{\mathsf{s}} (\,\overline{x : T}\,) \to D\,\overline{u}^n\ : \star \rightsquigarrow (\,\overline{x : \tau}\,) \to D\,\overline{u}^n}}}{\begin{array}{c}\Sigma \vdash_{\mathsf{d}} (\mathbf{data}\, D\,\overline{u : \kappa}^n\ =\ \overline{K\,\overline{x : T}}\,) : (D : (\,\overline{u : \kappa}^n\,) \to \star, \\ \overline{K : (\,\overline{u : \kappa}^n\,) \to (\,\overline{x : T}\,) \to D\,\overline{u}^n}\,) \rightsquigarrow e\end{array}}\ \textsc{TRdecl\_Data}$$

$$\begin{array}{ll} e \equiv \mathbf{let}\ D : (\,\overline{u : \rho}^n\,) \to \star = & \overline{\mu\, X : (\,\overline{u : \rho}^n\,) \to \star.\ \lambda\,\overline{u : \rho}^n\,.\ (b : \star) \to} \\ & \overline{((\,\overline{x : \tau[D \mapsto X]}\,) \to b)}\ \to b\ \mathbf{in} \\ \quad \mathbf{let}\ K_i : (\,\overline{u : \rho}^n\,) \to (\,\overline{x : \tau}\,) \to D\,\overline{u}^n & = \lambda\,\overline{u : \rho}^n\,.\ \lambda\,\overline{x : \tau}\,.\ \mathsf{cast}_{\uparrow}^{n+1}\,[D\,\overline{u}^n\,] \\ & (\lambda b : \star.\ \lambda\,\overline{c : (\,\overline{x : \tau}\,) \to b}\,.\ c_i\ \overline{x}\,)\ \mathbf{in} \end{array}$$

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

$$\frac{\begin{array}{c} K : (\,\overline{u : \kappa}^n\,) \to (\,\overline{x : T}\,) \to D\,\overline{u}^n\ \in\ \Sigma \\ \Sigma, \overline{x : T[\,\overline{u \mapsto v}\,]} \vdash_{\mathsf{s}} E : S \rightsquigarrow e \qquad \overline{\Sigma \vdash_{\mathsf{s}} T[\,\overline{u \mapsto v}\,] : \star \rightsquigarrow \tau'} \end{array}}{\Sigma \vdash_{\mathsf{p}} K\,\overline{x : T[\,\overline{u \mapsto v}\,]}\ \Rightarrow E : D\,\overline{v}^n\ \to S \rightsquigarrow \lambda\,\overline{x : \tau'}\,.\ e}\ \textsc{TRpat\_Alt}$$

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

$$\frac{\vdash_{\mathsf{wf}} \Sigma}{\Sigma \vdash_{\mathsf{s}} \star : \star \rightsquigarrow \star}\ \textsc{TR\_Ax}$$

$$\frac{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma \qquad x : T \in \Sigma}{\Sigma \vdash_{\mathsf{s}} x : T \rightsquigarrow x}\ \textsc{TR\_Var}$$

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

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

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

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

$$\frac{\Sigma \vdash_{\mathsf{s}} E_1 : D\,\overline{v}^n \rightsquigarrow e_1 \qquad \overline{\Sigma \vdash_{\mathsf{p}} p \Rightarrow E_2 : D\,\overline{v}^n\ \to S \rightsquigarrow e_2} \qquad \Sigma \vdash_{\mathsf{s}} S : \star \rightsquigarrow \sigma}{\Sigma \vdash_{\mathsf{s}} \mathbf{case}\, E_1\, \mathbf{of}\ \overline{p \Rightarrow E_2}\ : S \rightsquigarrow (\mathsf{cast}_{\downarrow}^{n+1}\ e_1)\,\sigma\ \overline{e_2}}\ \textsc{TR\_Case}$$

Figure 5. Type-directed translation rules of **Fun**

The rule states that $\lambda_\star^\mu$ expression e is the translation of the surface expression E of type T. The gray parts in Figure 5 define the translation rules.

Among others, Rules TRDECL_DATA, TRPAT_ALT and TR_CASE are of the essence to the translation. Rule TR_CASE translates case expressions into applications by first translating the scrutinee E_1 to e_1, casting it down to the right type. It is then applied to the result type of the body expression and a list of $\lambda_\star^\mu$ expressions of the alternatives. Rule TRPAT_ALT translates each alternative into a lambda abstraction, where each bound variable in the pattern corresponds to bound variables in the lambda abstraction in the same order. The body in the alternative is recursively translated and treated as the body of the lambda abstraction.

As for the translation of datatypes, we choose to work with the Scott encodings of datatypes [19]. Scott encodings encode case analysis, making it convenient to translate pattern matching. Rule TRDECL_DATA translates datatypes into $\lambda_\star^\mu$ expressions. First of all, it results in an incomplete expression (as can be seen by the incomplete let expressions). The result expression is supposed to be prepended to the translation of the last expression to form a complete $\lambda_\star^\mu$ expression, as specified by Rule TRPGM_PGM. Furthermore, each type constructor is translated to a recursive type, of which the body is a type-level lambda abstraction. What is interesting is that each recursive occurrence of the datatype in the data constructor parameters is replaced with the variable X. Note that for the moment, the result type variable b is restricted to have kind $\star$. This could pose difficulties when translating GADTs, which is an interesting topic for future work. Each data constructor is translated to a lambda abstraction. Notice how we use $\mathsf{cast}^\uparrow$ in the lambda body to get the right type.

The rest of the translation rules hold few surprises.

5.4 Type Safety of the Translation

Now that we have elaborated the translation, we can state the type safety theorem of the translation.

Theorem 4 (Type Safety of Expression Translation). *Given a surface language expression E and context Σ, if $\Sigma \vdash_{\mathsf{s}} E : T \rightsquigarrow e$, $\Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau$ and $\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma$, then $\Gamma \vdash e : \tau$.*

The proof is by induction on the derivation of $\Sigma \vdash_{\mathsf{s}} E : T \rightsquigarrow e$. Most of the rules are straightforward, following directly by induction. The most heavy one is the rule TR_CASE, where we combine the information from rule TRPGM_DATA and rule TRPAT_ALT with the operational semantics. The full proof of type-safety appears in the full version of this paper.

6 Related Work

In this paper, we give a positive answer as to whether we can find a calculus comparable in simplicity to PTS that models key features of modern functional languages (like Haskell and ML), while preserving type soundness and decidable type checking. To our knowledge we are the first to do so.

Core calculus for functional languages System F [25] is the simplest foundation for a polymorphic functional language. While the simplicity of System F is appealing, many features in use by functional languages, such as recursive types (which can be added in standard ways) or *higher-kinded polymorphism* (which requires major changes) are missing. $\lambda^{\mu}_{\star}$ is comparable in simplicity to System F, while being much more expressive. Girard's System F_{ω} [14] is a typed lambda calculus with higher-kinded polymorphism. Unlike System F, it introduces type operators (type-level functions). To guarantee the well-formedness of type expressions, an extra level of *kinds* is added to the system. In comparison, $\lambda^{\mu}_{\star}$ is considerably simpler than System F_{ω}, both in terms of language constructs and complexity of proofs. System F_{ω} differs from $\lambda^{\mu}_{\star}$ in that it uses a conversion rule due to its ability to perform type-level computation. $\lambda^{\mu}_{\star}$ can express type-computation in F_{ω}, but it requires explicit casts. Interestingly enough this feature of F_{ω} is actually not used by core languages for Haskell, because such core languages do not include type-level abstractions. Both System F_{ω} (with some standard extensions) and $\lambda^{\mu}_{\star}$ can express the key features required for the Haskell 2010 standard [18]. Nevertheless, System F_{ω} misses several features in use by modern GHC Haskell. In particular, features like kind polymorphism or datatype promotion would require non-trivial extensions to the system, complicating the system even more. Those features can be expressed directly in $\lambda^{\mu}_{\star}$.

$\lambda^{\mu}_{\star}$ still lacks support for some features of GHC Haskell. The current core language for GHC Haskell, System F_C [12] is a significant extension of System F_{ω}, which supports GADTs [21], functional dependencies [15], type families [12], and soon even kind equality [33]. These features use a non-trivial form of type equality, which is currently missing from $\lambda^{\mu}_{\star}$. On the other hand $\lambda^{\mu}_{\star}$ is rather simple, and has only 8 language constructs, whereas System F_C is significantly more complex and has currently over 30 language constructs. A primary goal of our future work is to investigate the addition of such forms of non-trivial type-equality. One key challenge is how to account for injectivity of type constructors, which is required for GADTs. Because datatypes are not built-in $\lambda^{\mu}_{\star}$, injectivity of type constructors requires a different approach from System F_C.

Casts for managed type-level computation Type-level computation in $\lambda^{\mu}_{\star}$ is controlled by explicit casts. Several studies [17, 26, 27, 29, 30] also attempt to use explicit casts for managed type-level computation. However, casts in those approaches require *equality proof terms*, while casts in $\lambda^{\mu}_{\star}$ do not. The need for equality proof terms complicates the language design because: 1) building equality proofs requires various other language constructs, adding to the complexity of the language design and metatheory; 2) It is desirable to ensure that the equality proofs are valid. Otherwise, one can easily build bogus equality proofs with non-termination, which could endanger type safety. To solve the later problem, Zombie [8, 28] contains a logical fragment as a consistent sub-language that guarantees equality proofs are valid. Other approaches include restricting valid equality proofs to be syntactic values only [26, 27], or having different languages for proofs and terms [17, 29].

Unified syntax with decidable type-checking Pure Type Systems (PTS) [4] show how a whole family of type systems can be implemented using just a single syntactic form. PTS are an obvious source of inspiration for our work. Although this paper presents a specific system based on λC, it should be easy to generalize $\lambda_\star^\mu$ in the same way as PTS and further show the applicability of our ideas to other systems. An early attempt of using a PTS-like syntax for an intermediate language for functional programming was Henk [22]. The Henk proposal was to use the *lambda cube* as a typed intermediate language, unifying all three levels. However the authors have not studied the addition of general recursion nor full dependent types.

Zombie [8, 28] is a dependently typed language using a single syntactic category. An interesting aspect of Zombie is that it is composed of two fragments: a logical fragment where every expression is known to terminate, and a programmatic fragment that allows general recursion. Even though Zombie has one syntactic category, it is still fairly complicated (with around 24 language constructs) as it tries to be both consistent as a logic and pragmatic as a programming language. The logical and programmatic fragments in Zombie are *tightly coupled*. Thus it's hard to remove language constructs even if we are only interested in modeling a programmatic fragment. For example, in Zombie, the conversion rule (TConv) for the programmatic part depends on equality proofs, which are only available in the logical part. In contrast to Zombie, $\lambda_\star^\mu$ takes another point of the design space, giving up logical consistency for simplicity in the language design.

Unified syntax with general recursion and undecidable type checking Cayenne [2] is a programming language that integrates the full power of dependent types with general recursion. It bears some similarities with $\lambda_\star^\mu$: Firstly, it also uses one syntactic form for both terms and types. Secondly, it allows arbitrary computation at type level. Thirdly, because of unrestricted recursion allowed in the system, Cayenne is logically inconsistent. However, the most crucial difference from $\lambda_\star^\mu$ is that type checking in Cayenne is undecidable. From a pragmatic point of view, this design choice simplifies the implementation, but the desirable property of decidable type checking is lost. Cardelli's Type:Type language [7] also features general recursion. He uses general recursion to implement equi-recursive types, thus unifying recursion and recursive types in a single construct. However, both equi-recursive types and the Type:Type axiom make the type system undecidable. In contrast $\lambda_\star^\mu$ generalizes iso-recursive types to control type-level computation and make type-checking decidable. $\Pi\Sigma$ [1] is another example of a language that uses one recursion mechanism for both types and functions, but it does not support decidable type checking either.

Restricted recursion with termination checking Dependently typed languages such as Coq [9] and Adga [20] are conservative as to what kind of computation is allowed. Coq, as a proof system, requires all programs to terminate by means of a termination checker, ensuring that recursive calls are only allowed on *syntactic subterms* of the primary argument. Thus decidable type checking, as well as logical consistency, are also preserved. The conservative, syntactic criteria is insufficient to support a variety of important programming paradigms. Agda and Idris [6], in addition, offer an option to disable the termination checker to allow for writing arbitrary functions. This, however, costs us both decidable type checking and logical consistency. While logical consistency is an appealing property, it is not a goal of $\lambda_\star^\mu$. Instead $\lambda_\star^\mu$ aims at retaining (term-level) general recursion as found in languages like Haskell or ML, while benefiting from a unified syntax to simplify the implementation of the core language.

Stratified type system with general recursion on term level One way to allow general recursion and dependent types in the same language and still have decidable type checking is to use multiple levels of syntax. In this way it is easy to have a term language with general recursion, but have a more restricted type and/or kind language. On the other hand this brings complexity to the language as multiple levels (possibly with similar constructs) have to be managed. $F^\star$ [31] is a recently proposed dependently typed language that supports writing general-purpose programs with effects while maintaining a consistent core language. In its core, it has several sub-languages – for terms, proofs and so on (more than 20 language constructs). In $F^\star$, the use of recursion is restricted in specifications and proofs, but arbitrary recursion is allowed in programs.

7 Conclusions and Future Work

This work proposes $\lambda_\star^\mu$: a minimal dependently typed core language that allows the same syntax for terms and types, supports type-level computation, and preserves decidable type checking under the presence of general recursion. The key idea is to control type-level computation using casts. Because each cast can only account for one-step of type-level computation, type checking becomes decidable without requiring strong normalization of the calculus. At the same time one-step casts together with recursion provide a generalization of iso-recursive types. By demonstrating a surface language, supporting advanced language constructs, on top of $\lambda_\star^\mu$ we have shown the potential of $\lambda_\star^\mu$ to serve as a core for Haskell-like languages.

In future work, we hope to make writing type-level computation easier by adding language constructs to the surface language. Currently intensive type-level computation can be written in $\lambda_\star^\mu$. However it is inconvenient to use, because in $\lambda_\star^\mu$ type-level computation is driven by casts, and the number of casts needs to be specified beforehand. Nevertheless, we do not consider this a critical weakness of our system. The design of $\lambda_\star^\mu$ is similar to System F_C which sacrifices the convenience of type-level computation and recovers the computation by surface-level language constructs, such as closed type families in Haskell. We plan to add new surface language constructs in the same spirit as type families in Haskell and automatically generate casts through the translation. We also hope to investigate how to add inductive families and GADTs to the surface language.

References

1. Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. *ΠΣ*: Dependent types without the sugar. In *Functional and Logic Programming*, pages 40–55. 2010.
2. Lennart Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.
3. Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 117–309, 1992.
4. HP Barendregt and HP Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1:124, 1991.
5. Richard Bird and Lambert Meertens. Nested datatypes. *Mathematics of program construction*, 1422:52–67, 1998.
6. Edwin Brady. IDRIS—systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, pages 43–54, 2011.
7. Luca Cardelli. *A Polymorphic lambda-calculus with Type: Type*. Digital Systems Research Center, 1986.
8. Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 33–45, 2014.
9. Coq Development Team. The *Coq* proof assistant. Documentation, system download. Contact: `http://coq.inria.fr/`.
10. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
11. Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.
12. Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.
13. Leonidas Fegaras and Tim 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*, pages 284–294, 1996.
14. Jean-Yves Girard. *Interprtation fonctionnelle et limination des coupures de l'arithmtique d'ordre suprieur*. PhD thesis, Universit Paris VII, 1972.
15. Mark P. Jones. Type Classes with Functional Dependencies. *Proceedings of the 9th European Symposium on Programming Languages and Systems*, (March), 2000.
16. LSV Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993.
17. Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *Sixth ACM SIGPLAN Workshop Programming Languages meets Program Verification (PLPV '12)*, pages 15–26, 2012.
18. Simon Marlow et al. Haskell 2010 language report. *Available online http://www.haskell. org/(May 2011)*, 2010.

19. T. Æ. Mogensen. Theoretical pearls: Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, 1992.
20. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
21. Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, July 2004.
22. S.L. Peyton Jones and E. Meijer. Henk: a Typed Intermediate Language. In *Types in Compilation Workshop*, 1997.
23. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, 1988.
24. Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.
25. John C. Reynolds. Towards a theory of type structure. In *Proceedings of the 'Colloque sur la Programmation'*, number 19, pages 408–425, Paris, France, 1974.
26. Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. Irrelevance, heterogenous equality, and call-by-value dependent type systems. In *Fourth workshop on Mathematically Structured Functional Programming (MSFP '12)*, pages 112–162, 2012.
27. Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 369–382, 2015.
28. Vilhem Sjöberg. *A Dependently Typed Language with Nontermination*. PhD thesis, University of Pennsylvania, 2015.
29. Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified programming in guru. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, pages 49–58. ACM, 2009.
30. Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the ACM SIGPLAN international workshop on Types in Languages Design and Implementation*, pages 53–66, 2007.
31. Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming*, pages 266–278, 2011.
32. Joseph Vanderwaart, Derek Dreyer, Leaf Petersen, Karl Crary, Robert Harper, and Perry Cheng. Typed compilation of recursive datatypes. *TLDI*, 38(3):98–108, 2003.
33. Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. Towards dependently typed Haskell: System FC with kind equality. In *International Conference on Functional Programming*, 2013.
34. Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66, 2012.

A Full Specification of Core Language

A.1 Syntax

$$
\begin{array}{llll}
e,\ \tau,\ \sigma,\ \rho & ::= & & \text{Expressions} \\
& | & x & \text{Variable} \\
& | & \star & \text{Type of Types} \\
& | & e_1\ e_2 & \text{Application} \\
& | & \lambda x : \tau.\ e & \text{Abstraction} \\
& | & \Pi\, x : \tau.\ \sigma & \text{Dependent Product} \\
& | & \mathsf{cast}^{\uparrow}\,[\tau]\ e & \text{Cast Up} \\
& | & \mathsf{cast}_{\downarrow}\ e & \text{Cast Down} \\
& | & \mu\, x : \tau.\ e & \text{Polymorphic Recursion}
\end{array}
$$

$$
\begin{array}{llll}
\Gamma & ::= & & \text{Contexts} \\
& | & \varnothing & \text{Empty} \\
& | & \Gamma, x : \tau & \text{Variable Binding}
\end{array}
$$

$$
\begin{array}{llll}
v & ::= & & \text{Values} \\
& | & \star & \text{Type of Types} \\
& | & \lambda x : \tau.\ e & \text{Abstraction} \\
& | & \Pi\, x : \tau_1.\ \tau_2 & \text{Dependent Product} \\
& | & \mathsf{cast}^{\uparrow}\,[\tau]\ e & \text{Cast Up}
\end{array}
$$

Syntactic Sugar

$$
\begin{array}{ll}
\tau_1 \to \tau_2 & \triangleq \Pi\, x : \tau_1.\ \tau_2,\ \text{ where } x \notin \mathsf{FV}(\tau_2) \\
(x : \tau_1) \to \tau_2 & \triangleq \Pi\, x : \tau_1.\ \tau_2 \\
\textbf{let}\ x : \tau = e_2\ \textbf{in}\ e_1 & \triangleq e_1[x \mapsto e_2] \\
\textbf{letrec}\ x : \tau = e_2\ \textbf{in}\ e_1 & \triangleq \textbf{let}\ x : \tau = \mu\, x : \tau.\ e_2\ \textbf{in}\ e_1 \\
\mathsf{cast}^{n}_{\uparrow}\,[\tau_1]\ e & \triangleq \mathsf{cast}^{\uparrow}[\tau_1](\mathsf{cast}^{\uparrow}[\tau_2](\ldots(\mathsf{cast}^{\uparrow}\,[\tau_n]\ e)\ldots)) \\
\mathsf{cast}^{n}_{\downarrow}\ e & \triangleq \underbrace{\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow}(\ldots(\mathsf{cast}_{\downarrow}}_{n}\ e)\ldots))
\end{array}
$$

A.2 Operational Semantics

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

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

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

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

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

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

A.3 Typing

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

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

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

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \star : \star} \quad \textsc{T\_Ax}$$

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

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

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

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

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

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

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

B Proofs about Core Language

B.1 Properties

We follow the naming of lemmas and proofs of properties for Pure Type System from [3]. Some lemmas have other well-known names, like Lemma 3 is often called *Weakening* and Lemma 5 is often called *Inversion*.

Lemma 3 (Thinning). *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 $\Gamma \vdash e : \tau$. $\square$

Lemma 4 (Substitution). *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 $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$. We use the notation $e^* \equiv e[x \mapsto e_2]$ to denote the substitution for short. 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 since other cases can be easily followed by the proof for PTS in [3]. Consider the last step of derivation of the following cases:

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

By the induction hypothesis, we have $\Gamma_1, \Gamma_2^*, y : \tau^* \vdash {e_1}^* : \tau^*$ and $\Gamma_1, \Gamma_2^* \vdash \tau^* : \star$. Then by the derivation rule, $\Gamma_1, \Gamma_2^* \vdash (\mu\, y : \tau^*.\ {e_1}^*) : \tau^*$. Thus we can conclude $\Gamma_1, \Gamma_2^* \vdash (\mu\, y : \tau.\ e_1)^* : \tau^*$.

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 the 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 derivation rule, $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}^{\uparrow}\, [\tau_1^*]\, {e_1}^*) : \tau_1^*$. Thus we can conclude $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}^{\uparrow}\, [\tau_1]\, e_1)^* : \tau_1^*$.

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 the 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 derivation rule, $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}_{\downarrow}\, {e_1}^*) : \tau_2^*$. Thus we can conclude $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}_{\downarrow}\, e_1)^* : \tau_2^*$.

□

Lemma 5 (Generation). *If the alpha equivalence is witnessed by notation* $\equiv$*, we have the following results:*

(1) If $\Gamma \vdash x : \sigma$*, then there exists 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* $\Gamma \vdash e_1 : (\Pi\, x : \tau_2.\ \tau_1)$*,* $\Gamma \vdash e_2 : \tau_2$ *and* $\sigma \equiv \tau_1[x \mapsto e_2]$*.*
(3) If $\Gamma \vdash (\lambda x : \tau_1.\ e) : \sigma$*, then there exists 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 exists 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 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 3, it can be shown that the statement of the lemma holds and is the only possible case. $\square$

Lemma 6 (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 5. $\square$

B.2 Decidability of Type Checking

Lemma 7 (Decidability of One-step Reduction). *The one-step reduction $\longrightarrow$ is called decidable if given e there is a unique e' such that $e \longrightarrow e'$ or there is no such e'.*

Proof. By induction on the structure of e:

Case $e = x$: e is a variable which does not match any rules of $\longrightarrow$. Thus there is no e' such that $e \longrightarrow e'$.

Case $e = v$: e is a value that has one of the following forms: (1) $\star$, (2) $\lambda x : \tau.\ e$, (3) $\Pi\, x : \tau_1.\ \tau_2$, (4) $\mathsf{cast}^{\uparrow}\,[\tau]\ e$. Thus, it does not match any rules of $\longrightarrow$. Then there is no e' such that $e \longrightarrow e'$.

Case $e = (\lambda x : \tau.\ e_1)\ e_2$: Since the first term $\lambda x : \tau.\ e_1$ is a value, rule S_App does not apply to this case. Thus, only rule S_Beta can be applied and there is a unique $e' = e_1[x \mapsto e_2]$.

Case $e = \mathsf{cast}_{\downarrow}\,(\mathsf{cast}^{\uparrow}\,[\tau]\ e_1)$: Since the inner term $\mathsf{cast}^{\uparrow}\,[\tau]\ e_1$ is a value, rule S_CastDown does not apply to this case. Thus, only rule S_CastDownUp can be applied and there is a unique $e' = e_1$.

Case $e = \mu\, x : \tau.\ e_1$: Only rule S_Mu can be applied. Thus, there is a unique $e' = e_1[x \mapsto \mu\, x : \tau.\ e_1]$.

Case $e = e_1\ e_2$ and e_1 is not a λ-term: If $e_1 = v$ and is not a λ-term, there is no rule to reduce e. Then there is no e_1' such that $e_1 \longrightarrow e_1'$, which does not satisfy the premise of rule S_App. Thus, there is no e' such that $e \longrightarrow e'$. Otherwise, if e_1 is not a value, there exists some e_1' such that $e_1 \longrightarrow e_1'$. By the induction hypothesis, e_1' is the unique reduction of e_1. Thus, by rule S_App, $e' = e_1'\ e_2$ is the unique reduction of e.

Case $e = \mathsf{cast}_{\downarrow}\, e_1$ **and** e_1 **is not a** $\mathsf{cast}^{\uparrow}$**-term:** If $e_1 = v$ and is not a $\mathsf{cast}^{\uparrow}$-term, there is no rule to reduce e. Then there is no e_1' such that $e_1 \longrightarrow e_1'$, which does not satisfy the premise of rule S_CastDown. Thus, there is no e' such that $e \longrightarrow e'$.

Otherwise, if e_1 is not a value, there exists some e_1' such that $e_1 \longrightarrow e_1'$. By the induction hypothesis, e_1' is the unique reduction of e_1. Thus, by rule S_CastDown, $e' = \mathsf{cast}_{\downarrow}\, e_1'$ is the unique reduction of e.

□

Theorem 5 (Decidability of Type Checking). *There is an 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. If $x : \tau \in \Gamma$, then τ is the unique type of x such that $\Gamma \vdash x : \tau$. Otherwise, if $x \notin \mathsf{dom}(\Gamma)$, there is no such τ.

Case $e = e_1\, e_2$**:** By rule T_App and induction hypothesis, there exist unique τ_1 and τ_2 such that $\Gamma \vdash e_1 : (\Pi\, x : \tau_1.\ \tau_2)$, $\Gamma \vdash e_2 : \tau_1$. Thus, $\tau_2[x \mapsto e_2]$ is the unique type of e such that $\Gamma \vdash e : \tau_2[x \mapsto e_2]$.

Case $e = \lambda x : \tau_1.\ e_1$**:** By rule T_Lam and induction hypothesis, there exist unique τ_2 such that $\Gamma \vdash (\Pi\, x : \tau_1.\ \tau_2) : \star$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$. Thus, $\Pi\, x : \tau_1.\ \tau_2$ is the unique type of e such that $\Gamma \vdash e : \Pi\, x : \tau_1.\ \tau_2$.

Case $e = \Pi\, x : \tau_1.\ \tau_2$**:** By rule T_Pi and induction hypothesis, we have $\Gamma \vdash \tau_1 : \star$ and $\Gamma, x : \tau_1 \vdash \tau_2 : \star$. Thus, $\star$ is the unique type of e such that $\Gamma \vdash e : \star$.

Case $e = \mu\, x : \tau.\ e_1$**:** By rule T_Mu and induction hypothesis, we have $\Gamma \vdash \tau : \star$ and $\Gamma, x : \tau \vdash e : \tau$. Thus, τ is the unique type of e such that $\Gamma \vdash e : \tau$.

Case $e = \mathsf{cast}^{\uparrow}\, [\tau_1]\, e_1$**:** From the premises of rule T_CastUp, by the induction hypothesis, we can derive the type of e_1 as τ_2 by $\Gamma \vdash e_1 : \tau_2$, and check whether τ_1 is legal by $\Gamma \vdash \tau_1 : \star$. For a legal τ_1, by Lemma 7, there is a unique τ_1' such that $\tau_1 \longrightarrow \tau_1'$ or there is no such τ_1'. If such τ_1' does not exist, then we report type checking fails.

Otherwise, we examine if τ_1' is syntactically equal to τ_2, i.e., $\tau_1' \equiv \tau_2$. If the equality holds, we conclude the unique type of e is τ_1, i.e., $\Gamma \vdash e : \tau_1$. Otherwise, we report e fails to type check.

Case $e = \mathsf{cast}_{\downarrow}\, e_1$**:** From the premises of rule T_CastDown, by the induction hypothesis, we can derive the type of e_1 as τ_1 by $\Gamma \vdash e_1 : \tau_1$. By Lemma 7, there is a unique τ_2 such that $\tau_1 \longrightarrow \tau_2$ or such τ_2 does not exist.

If such τ_2 exists and its sorts is $\star$, we find the unique type of e is τ_2 and can conclude $\Gamma \vdash e : \tau_2$. Otherwise, we report e fails to type check.

□

B.3 Type Safety

Definition 2 (Multi-step reduction). *The relation $\twoheadrightarrow$ is the transitive and reflexive closure of $\longrightarrow$.*

Definition 3 **(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.*

Theorem 6 **(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 theorem follows 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 5(2), there exist expressions τ_1' and τ_2 such that

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

By Lemma 5(3), the judgement (1) implies that there exists an expression τ_2' such that

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

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 4, 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 5(2), there exist expressions τ_1 and τ_2 such that

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

By the 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 5(7), there exist expressions τ_1, τ_2 such that

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

By the 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 5(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 5(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 7 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 5(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 4, 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, $\sigma' \equiv \tau[x \mapsto \mu\, x : \tau.\ e] \equiv \tau \equiv \sigma$.

$\square$

Theorem 7 (Progress). *If $\varnothing \vdash e : \sigma$ then either e is a value v or there exists e' such that $e \longrightarrow e'$.*

Proof. By induction on the derivation of $\varnothing \vdash e : \sigma$ 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 5(2), there exist expressions τ_1 and τ_2 such that $\varnothing \vdash e_1 : (\Pi\, x : \tau_1.\ \tau_2)$ and $\varnothing \vdash e_2 : \tau_1$. Consider whether e_1 is a value:

- If $e_1 = v$, by Lemma 5(3), it must be a λ-term such that $e_1 \equiv \lambda x : \tau_1.\ e_1'$ for some e_1' satisfying $\varnothing \vdash e_1' : \tau_2$. 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 the 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 5(7), there exist expressions τ_1 and τ_2 such that $\varnothing \vdash e_1 : \tau_1$ and $\tau_1 \longrightarrow \tau_2$. Consider whether e_1 is a value:

- If $e_1 = v$, by Lemma 5(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 $\varnothing \vdash e_1' : \tau_2$. 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 the 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 Surface Language

C.1 Syntax

See Figure 6.

C.2 Expression Typing

See Figure 7.

C.3 Translation to the Core

See Figure 8.

D Proofs about Surface Language

D.1 Type Safety of the Translation

Theorem 8 (Type Safety of Expression Translation). *Given a surface language expression E and context Σ, if $\Sigma \vdash_{\mathsf{s}} E : T \rightsquigarrow e$, $\Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau$ and $\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma$, then $\Gamma \vdash e : \tau$.*

Proof. By induction on the derivation of $\Sigma \vdash_{\mathsf{s}} E : T \rightsquigarrow e$. Suppose there is a core language context Γ such that $\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma$.

Case $\dfrac{\vdash_{\mathsf{wf}} \Sigma}{\Sigma \vdash_{\mathsf{s}} \star : \star \rightsquigarrow \star}$ **TR_Ax:**

Trivial. $e = \tau = \star$ and $\Gamma \vdash \star : \star$ holds by rule T_Ax.

Case $\dfrac{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma \qquad x : T \in \Sigma}{\Sigma \vdash_{\mathsf{s}} x : T \rightsquigarrow x}$ **TR_Var:**

Trivial. By rule T_Var, we have $\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma$, then $x : \tau \in \Gamma$ where $\Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau$.

$$
\begin{array}{llll}
pgm & ::= & & \text{Program} \\
 & | & \overline{decl}\,; E & \ \text{Declarations} \\
decl & ::= & & \text{Declarations} \\
 & | & \mathbf{data}\, D\, \overline{u : \kappa} \ = \ \overline{K\, \overline{x : T}} & \ \text{Datatype} \\
u,\ \upsilon & ::= & & \text{Atoms} \\
 & | & x & \ \text{Variable} \\
 & | & K & \ \text{Data Constructor} \\
p & ::= & & \text{Patterns} \\
 & | & K\, \overline{x : T} & \ \text{Pattern} \\
E,\ T,\ S,\ \kappa & ::= & & \text{Expressions} \\
 & | & u & \ \text{Atom} \\
 & | & \star & \ \text{Type of Types} \\
 & | & E_1\, E_2 & \ \text{Application} \\
 & | & \lambda x : T.\ E & \ \text{Abstraction} \\
 & | & \Pi\, x : T.\ S & \ \text{Dependent Product} \\
 & | & \mu\, x : T.\ E & \ \text{Recursion} \\
 & | & \mathbf{case}\, E_1\, \mathbf{of}\ \overline{p \Rightarrow E_2} & \ \text{Case Analysis} \\
\Sigma & ::= & & \text{Contexts} \\
 & | & \varnothing & \ \text{Empty} \\
 & | & \Sigma, u : T & \ \text{Atom Binding}
\end{array}
$$

Syntactic Sugar

$$
\begin{array}{l}
\mathbf{rcrd}\, R\, \overline{u : \kappa}^n \ = K\{\, \overline{N : T}\,\} \triangleq \mathbf{data}\, R\, \overline{u : \kappa}^n \ = K\, \overline{x : T}\,; \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ N_i : (\, \overline{u : \kappa}^n\,) \to R\, \overline{u}^n \ \to T_i = \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda\, \overline{u : \kappa}^n \,.\, \lambda y : R\, \overline{u}^n \,. \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{case}\, y\, \mathbf{of}\ K\, \overline{x : T} \ \Rightarrow x_i\ \mathbf{in}
\end{array}
$$

Figure 6. Syntax of the surface language

Case $\dfrac{\Sigma \vdash_{\mathsf{s}} E_1 : (\Pi\, x : T_2.\ T_1) \rightsquigarrow e_1 \qquad \Sigma \vdash_{\mathsf{s}} E_2 : T_2 \rightsquigarrow e_2}{\Sigma \vdash_{\mathsf{s}} E_1\, E_2 : T_1[x \mapsto E_2] \rightsquigarrow e_1\, e_2}$ **TR_App:**

Suppose

$$\begin{array}{l} \Sigma \vdash_{\mathsf{s}} E_1\, E_2 : T_1[x \mapsto E_2] \rightsquigarrow e_1\, e_2 \\ \Sigma \vdash_{\mathsf{s}} T_1[x \mapsto E_2] : \star \rightsquigarrow \tau_1[x \mapsto e_2]\,. \end{array}$$

By induction hypothesis, we have $\Gamma \vdash e_1 : (\Pi\, x : \tau_2.\ \tau_1), \Gamma \vdash e_2 : \tau_2$, where

$$\begin{array}{l} \Sigma \vdash_{\mathsf{s}} E_1 : (\Pi\, x : T_2.\ T_1) \rightsquigarrow e_1 \\ \Sigma \vdash_{\mathsf{s}} (\Pi\, x : T_2.\ T_1) : \star \rightsquigarrow (\Pi\, x : \tau_2.\ \tau_1) \\ \Sigma \vdash_{\mathsf{s}} E_2 : T_2 \rightsquigarrow e_2 \\ \Sigma \vdash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2. \end{array}$$

Thus by rule T_App, we can conclude $\Gamma \vdash e_1\, e_2 : \tau_1[x \mapsto e_2]\,$.

Case $\dfrac{\Sigma, x : T_1 \vdash_{\mathsf{s}} E : T_2 \rightsquigarrow e \qquad \Sigma \vdash_{\mathsf{s}} (\Pi\, x : T_1.\ T_2) : \star \rightsquigarrow \Pi\, x : \tau_1.\ \tau_2}{\Sigma \vdash_{\mathsf{s}} (\lambda x : T_1.\ E) : (\Pi\, x : T_1.\ T_2) \rightsquigarrow \lambda x : \tau_1.\ e}$ **TR_Lam:**

Suppose

$$\begin{array}{l} \Sigma \vdash_{\mathsf{s}} (\lambda x : T_1.\ E) : (\Pi\, x : T_1.\ T_2) \rightsquigarrow \lambda x : \tau_1.\ e \\ \Sigma \vdash_{\mathsf{s}} \Pi\, x : T_1.\ T_2 : \star \rightsquigarrow \Pi\, x : \tau_1.\ \tau_2. \end{array}$$

By the induction hypothesis, we have $\Gamma, x : \tau_1 \vdash e : \tau_2, \Gamma \vdash \Pi\, x : \tau_1.\ \tau_2 : \star$ where

$$\begin{array}{ll} \Sigma, x : T_1 \vdash_{\mathsf{s}} E : T_2 \rightsquigarrow e & \\ \Sigma \vdash_{\mathsf{s}} T_1 : \star \rightsquigarrow \tau_1 & \Sigma \vdash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2 \\ \Sigma \vdash_{\mathsf{s}} (\Pi\, x : T_1.\ T_2) : \star \rightsquigarrow \Pi\, x : \tau_1.\ \tau_2 & \end{array}$$

Thus by rule T_Lam, we can conclude $\Gamma \vdash (\lambda x : \tau_1.\ e) : (\Pi\, x : \tau_1.\ \tau_2)$.

Case $\dfrac{\Sigma \vdash_{\mathsf{s}} T_1 : \star \rightsquigarrow \tau_1 \qquad \Sigma, x : T_1 \vdash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2}{\Sigma \vdash_{\mathsf{s}} (\Pi\, x : T_1.\ T_2) : \star \rightsquigarrow \Pi\, x : \tau_1.\ \tau_2}$ **TR_Pi:**

Suppose

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

By the induction hypothesis, we have $\Gamma \vdash \tau_1 : \star, \Gamma, x : \tau_1 \vdash \tau_2 : \star$ where $\Sigma \vdash_{\mathsf{s}} T_1 : \star \rightsquigarrow \tau_1, \Sigma, x : T_1 \vdash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2$ Thus by rule T_Pi we can conclude $\Gamma \vdash (\Pi\, x : \tau_1.\ \tau_2) : \star$.

Case $\dfrac{\Sigma, x : T \vdash_{\mathsf{s}} E : T \rightsquigarrow e \qquad \Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau}{\Sigma \vdash_{\mathsf{s}} (\mu\, x : T.\ E) : T \rightsquigarrow \mu\, x : \tau.\ e}$ **TR_Mu:**

Suppose

$$\begin{array}{l} \Sigma \vdash_{\mathsf{s}} (\mu\, x : T.\ E) : T \rightsquigarrow \mu\, x : \tau.\ e \\ \Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau. \end{array}$$

By the induction hypothesis, we have

$$\Gamma, x : \tau \vdash e : \tau, \text{ where } \Sigma, x : T \vdash_{\mathsf{s}} E : T \rightsquigarrow e.$$

Thus by rule T_Mu, we can conclude $\Gamma \vdash (\mu\, x : \tau.\ e) : \tau$.

$$\textbf{Case}\ \frac{\dfrac{\Sigma \vdash_{\mathsf{s}} E_1 : D\,\overline{\upsilon}^n \rightsquigarrow e_1}{\overline{\Sigma \vdash_{\mathsf{p}} p \Rightarrow E_2 : D\,\overline{\upsilon}^n \to S \rightsquigarrow e_2}} \qquad \Sigma \vdash_{\mathsf{s}} S : \star \rightsquigarrow \sigma}{\Sigma \vdash_{\mathsf{s}} \textbf{case}\, E_1\, \textbf{of}\ \overline{p \Rightarrow E_2} : S \rightsquigarrow (\mathsf{cast}_{\downarrow}^{n+1}\, e_1)\, \sigma\ \overline{e_2}}\quad \textbf{TR\_Case:}$$

Suppose

$$\begin{array}{l} \Sigma \vdash_{\mathsf{s}} \textbf{case}\, E_1\, \textbf{of}\ \overline{p \Rightarrow E_2} : S \rightsquigarrow (\mathsf{cast}_{\downarrow}^{n+1}\, e_1)\, \sigma\ \overline{e_2} \\ \Sigma \vdash_{\mathsf{s}} S : \star \rightsquigarrow \sigma. \end{array}$$

By the induction hypothesis, we have

$$\begin{array}{ll} \Sigma \vdash_{\mathsf{s}} E_1 : D\,\overline{\upsilon}^n \rightsquigarrow e_1 & \Sigma \vdash_{\mathsf{s}} D\,\overline{\upsilon}^n : \star \rightsquigarrow \tau_1 \\ \Gamma \vdash e_1 : \tau_1 & \overline{\Sigma \vdash_{\mathsf{p}} p \Rightarrow E_2 : D\,\overline{\upsilon}^n \to S \rightsquigarrow e_2} \end{array}$$

By rule TRpat_Alt, we have

$$\begin{aligned} p &\equiv K\,\overline{x : T[\,\overline{u \mapsto \upsilon}\,]} \\ \overline{e_2} &\equiv \overline{\lambda\,\overline{x : \tau'}\,.\ e} \end{aligned}$$

where

$$\begin{array}{ll} \overline{\Sigma \vdash_{\mathsf{s}} E_2 : S \rightsquigarrow e} & \overline{\Gamma \vdash e : \sigma} \\ \overline{\Sigma \vdash_{\mathsf{s}} \upsilon : \star \rightsquigarrow u'} & \overline{\Sigma \vdash_{\mathsf{s}} T[\,\overline{u \mapsto \upsilon}\,] : \star \rightsquigarrow \tau[\,\overline{u \mapsto u'}\,]} \\ \tau' \equiv \tau[\,\overline{u \mapsto u'}\,] & \end{array}$$

By rule TRdecl_Data, we have $D \equiv \mu X : (\,\overline{u : \rho}^n\,) \to \star.\ \lambda\,\overline{u : \rho}^n\,.\ (b : \star) \to \overline{((\,\overline{x : \tau[D \mapsto X]}\,) \to b)} \to b$. Thus,

$$\tau_1 \equiv D\,\overline{u'}^{\,n},\ \text{where}\ \overline{\Gamma \vdash u' : \rho}\,.$$

Note that by operational semantics of $\lambda_\star^\mu$, the following reduction sequence follows for τ_1:

$$\begin{aligned} D\,\overline{u'}^{\,n} &\longrightarrow (\lambda\,\overline{u : \rho}^n\,.\ (b : \star) \to \overline{((\,\overline{x : \tau[D \mapsto X]\,[X \mapsto D]}\,) \to b)} \to b)\,\overline{u'}^{\,n} \\ &\longrightarrow_n (b : \star) \to \overline{(\,\overline{x : \tau'}\,) \to b} \to b \end{aligned}$$

Then by rule T_CastDown and the definition of n-step cast operator, the type of $\mathsf{cast}_{\downarrow}^{n+1}\, e_1$ is

$$(b : \star) \to \overline{(\,\overline{x : \tau'}\,) \to b} \to b.$$

Note that by rule T_Lam, $\Gamma \vdash e_2 : (\,\overline{x : \tau'}\,) \to \sigma$. Therefore, by rule T_App, we can conclude $\Gamma \vdash (\mathsf{cast}_{\downarrow}^{n+1}\, e_1)\, \sigma\ \overline{e_2} : \sigma$.

$\square$

$\boxed{\vdash_{\mathsf{wf}} \Sigma}$ Context well-formedness

$$\frac{}{\vdash_{\mathsf{wf}} \varnothing}\ \textsc{TSenv\_Empty}$$

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

$\boxed{\Sigma \vdash_{\mathsf{pg}} \mathit{pgm} : T}$ Program typing

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

$\boxed{\Sigma \vdash_{\mathsf{d}} \mathit{decl} : \Sigma'}$ Datatype typing

$$\frac{\begin{array}{c}\Sigma \vdash_{\mathsf{s}} (\,\overline{u : \kappa}^{n}\,) \to \star : \star \\ \hline \overline{\Sigma, D : (\,\overline{u : \kappa}^{n}\,) \to \star, \overline{u : \kappa}^{n} \vdash_{\mathsf{s}} (\,\overline{x : T}\,) \to D\,\overline{u}^{n} \;: \star}\end{array}}{\begin{array}{c}\Sigma \vdash_{\mathsf{d}} (\mathbf{data}\, D\, \overline{u : \kappa}^{n} \;=\; \overline{K\, \overline{x : T}}\,) : (D : (\,\overline{u : \kappa}^{n}\,) \to \star, \\ \overline{K : (\,\overline{u : \kappa}^{n}\,) \to (\,\overline{x : T}\,) \to D\,\overline{u}^{n}}\,)\end{array}}\ \textsc{TSdecl\_Data}$$

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

$$\frac{\begin{array}{c}K : (\,\overline{u : \kappa}^{n}\,) \to (\,\overline{x : T}\,) \to D\,\overline{u}^{n} \;\in \Sigma \\ \hline \Sigma, \overline{x : T[\,\overline{u \mapsto v}\,]} \vdash_{\mathsf{s}} E : S \qquad \overline{\Sigma \vdash_{\mathsf{s}} T[\,\overline{u \mapsto v}\,] \;: \star}\end{array}}{\Sigma \vdash_{\mathsf{p}} K\, \overline{x : T[\,\overline{u \mapsto v}\,]} \;\Rightarrow E : D\,\overline{v}^{n} \;\to S}\ \textsc{TSpat\_Alt}$$

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

$$\frac{\vdash_{\mathsf{wf}} \Sigma}{\Sigma \vdash_{\mathsf{s}} \star : \star}\ \textsc{TS\_Ax}$$

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

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

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

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

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

$$\frac{\Sigma \vdash_{\mathsf{s}} E_1 : D\,\overline{v}^{n} \qquad \overline{\Sigma \vdash_{\mathsf{p}} p \Rightarrow E_2 : D\,\overline{v}^{n} \;\to S} \qquad \Sigma \vdash_{\mathsf{s}} S : \star}{\Sigma \vdash_{\mathsf{s}} \mathbf{case}\, E_1\, \mathbf{of}\ \overline{p \Rightarrow E_2} \;: S}\ \textsc{TS\_Case}$$

Figure 7. Typing rules of the surface language

$\boxed{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma}$ Context well-formedness

$$\frac{}{\vdash_{\mathsf{wf}} \varnothing \rightsquigarrow \varnothing}\ \textsc{TRenv\_Empty}$$

$$\frac{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma \qquad \Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau}{\vdash_{\mathsf{wf}} \Sigma, x : T \rightsquigarrow \Gamma, x : \tau}\ \textsc{TRenv\_Var}$$

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

$$\frac{\overline{\Sigma_0 \vdash_{\mathsf{d}} decl : \Sigma' \rightsquigarrow e_1} \qquad \Sigma = \Sigma_0, \overline{\Sigma'} \qquad \Sigma \vdash_{\mathsf{s}} E : T \rightsquigarrow e}{\Sigma_0 \vdash_{\mathsf{pg}} (\,\overline{decl}\,; E) : T \rightsquigarrow \overline{e_1} \uplus e}\ \textsc{TRpgm\_Pgm}$$

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

$$\frac{\dfrac{\Sigma \vdash_{\mathsf{s}} (\,\overline{u : \kappa}^n\,) \to \star : \star \rightsquigarrow (\,\overline{u : \rho}^n\,) \to \star}{\overline{\Sigma, D : (\,\overline{u : \kappa}^n\,) \to \star, \overline{u : \kappa}^n \vdash_{\mathsf{s}} (\,\overline{x : T}\,) \to D\,\overline{u}^n\ : \star \rightsquigarrow (\,\overline{x : \tau}\,) \to D\,\overline{u}^n}}}{\begin{array}{c}\Sigma \vdash_{\mathsf{d}} (\mathbf{data}\,D\,\overline{u : \kappa}^n\ =\ \overline{K\,\overline{x : T}}\,) : (D : (\,\overline{u : \kappa}^n\,) \to \star,\\ \overline{K : (\,\overline{u : \kappa}^n\,) \to (\,\overline{x : T}\,) \to D\,\overline{u}^n}\,) \rightsquigarrow e\end{array}}\ \textsc{TRdecl\_Data}$$

$$\begin{array}{ll} e \equiv \mathbf{let}\ D : (\,\overline{u : \rho}^n\,) \to \star = & \mu\,X : (\,\overline{u : \rho}^n\,) \to \star.\ \lambda\,\overline{u : \rho}^n\,.\ (b : \star) \to \\ & \overline{((\,\overline{x : \tau[D \mapsto X]}\,) \to b)} \to b\ \mathbf{in} \\ \quad \mathbf{let}\ K_i : (\,\overline{u : \rho}^n\,) \to (\,\overline{x : \tau}\,) \to D\,\overline{u}^n & = \lambda\,\overline{u : \rho}^n\,.\ \lambda\,\overline{x : \tau}\,.\ \mathsf{cast}_{\uparrow}^{n+1}\,[D\,\overline{u}^n\,] \\ & (\lambda b : \star.\ \lambda\,\overline{c : (\,\overline{x : \tau}\,) \to b}\,.\ c_i\ \overline{x}\,)\ \mathbf{in} \end{array}$$

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

$$\frac{\begin{array}{c} K : (\,\overline{u : \kappa}^n\,) \to (\,\overline{x : T}\,) \to D\,\overline{u}^n\ \in \Sigma \\ \Sigma, \overline{x : T[\,\overline{u \mapsto v}\,]}\ \vdash_{\mathsf{s}} E : S \rightsquigarrow e \qquad \overline{\Sigma \vdash_{\mathsf{s}} T[\,\overline{u \mapsto v}\,]\ : \star \rightsquigarrow \tau'} \end{array}}{\Sigma \vdash_{\mathsf{p}} K\,\overline{x : T[\,\overline{u \mapsto v}\,]}\ \Rightarrow E : D\,\overline{v}^n\ \to S \rightsquigarrow \lambda\,\overline{x : \tau'}\,.\ e}\ \textsc{TRpat\_Alt}$$

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

$$\frac{\vdash_{\mathsf{wf}} \Sigma}{\Sigma \vdash_{\mathsf{s}} \star : \star \rightsquigarrow \star}\ \textsc{TR\_Ax}$$

$$\frac{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma \qquad x : T\ \in \Sigma}{\Sigma \vdash_{\mathsf{s}} x : T \rightsquigarrow x}\ \textsc{TR\_Var}$$

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

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

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

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

$$\frac{\Sigma \vdash_{\mathsf{s}} E_1 : D\,\overline{v}^n \rightsquigarrow e_1 \qquad \overline{\Sigma \vdash_{\mathsf{p}} p \Rightarrow E_2 : D\,\overline{v}^n\ \to S \rightsquigarrow e_2} \qquad \Sigma \vdash_{\mathsf{s}} S : \star \rightsquigarrow \sigma}{\Sigma \vdash_{\mathsf{s}} \mathbf{case}\ E_1\ \mathbf{of}\ \overline{p \Rightarrow E_2}\ : S \rightsquigarrow (\mathsf{cast}_{\downarrow}^{n+1}\ e_1)\ \sigma\ \overline{e_2}}\ \textsc{TR\_Case}$$

Figure 8. Translation rules of the surface language