Type-Level Computation One Step at a Time

Foo Bar Baz

The University of Foo

{foo,bar,baz}@foo.edu

Abstract

Many type systems support a conversion rule that allows type-level computation. In such type systems ensuring the *decidability* of type checking requires type-level computation to terminate. For calculi where the syntax of types and terms is the same, the decidability of type-checking is usually dependent on the strong normalization of the calculus, which ensures termination. An unfortunate consequence of this coupling between decidability and strong normalization is that adding (unrestricted) general recursion to such calculi is not possible.

This paper proposes an alternative to the conversion rule that allows the same syntax for types and terms, type-level computation, and preserves decidability of type-checking under the presence of general recursion. The key idea, which is inspired by the traditional treatment of *iso-recursive types*, is to make each type-level computation step explicit. Each beta reduction or expansion at the type-level is introduced by a language construct. This allows control over the type-level computation and ensures decidability of type-checking even in the presence of non-terminating programs at the type-level. We realize this idea by presenting a variant of the calculus of constructions with general recursion and recursive types. Furthermore we show how many advanced programming language features of state-of-the-art functional languages (such as Haskell) can be encoded in our minimalistic core calculus.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Design

Keywords Dependent types, Intermediate langauge

1. Introduction

Modern statically typed functional languages (such as ML, Haskell, Scala or OCaml) have increasingly expressive type systems. Often these large source languages are translated into a much smaller typed core language. The choice of the core language is essential to ensure that all the features of the source language can be encoded. For a simple polymorphic functional language it is possible, for example, to pick a variant of System F as a core language. However, the desire for more expressive type system features puts pressure on the core languages, often requiring them to be extended to support new features. For example, if the source language supports *higher-kinded types* or *type-level functions* then System F is not expressive enough and can no longer be used as the core language. Instead another core language that does provide support for higher-kinded types, such as System F_ω, needs to be used. However System F_ω is significantly more complex than System F and thus harder to maintain. If later a new feature, such as *kind polymorphism*, is desired the core language may need to be changed again to account for the new feature, introducing at the same time new sources of complexity. Indeed the core language for modern versions of functional languages are quite complex, having multiple syntactic sorts (such as terms, types and kinds), as well as dozens of language constructs []BRUNO: F_C.

The more expressive type systems become, the more types become similar to the terms. Therefore a natural idea is to unify terms and types. There are obvious benefits in this approach: only one syntactic level (terms) is needed; and there are much less language constructs, making the core language easier to implement and maintain. At the same time the core language becomes more expressive, giving us for free many useful language features. *Pure type systems* [] build on this observation and they show how a whole family of type systems (including System F and System F_ω) can be implemented using just a single syntactic form. With the added expressiveness it is even possible to have type-level programs expressed using the same syntax as terms as well as dependently typed programs [].

However having the same syntax for types and terms can also be problematic. If arbitrary type-level computation is allowed then type-level programs can use the same language constructs as terms. Usually type systems have a conversion rule to support type-level computation. In such type systems ensuring the *decidability* of type checking requires type-level computation to terminate. For calculi where the syntax of types and terms is the same, the decidability of type-checking is usually dependent on the strong normalization of the calculus, which ensures termination. An unfortunate consequence of this coupling between decidability and strong normalization is that adding (unrestricted) general recursion to such calculi is not possible. There is a clear tension between decidability of type-checking and allowing general recursion at the type-level.

This paper proposes $\lambda_\star^\mu$: a variant of the calculus of constructions allows the same syntax for types and terms, supports type-level computation, and preserves decidability of type-checking under the presence of general recursion. In $\lambda_\star^\mu$, each type-level computation step is explicit. BRUNO: emphasis on the advantages: a minimal core language? The key idea, which is inspired by the traditional treatment of *iso-recursive types*, is to introduce each beta reduction or expansion at the type-level by a *type-safe cast*. The casts allow control over the type-level computation. For example, if a type-level program requires two beta-reductions to reach normal form, then two casts are needed in the program. If a non-terminating program is used at the type-level, it is not possible to cause non-termination in the type-checker, because that would require a program with an infinite number of casts. Therefore, since single beta-steps are trivially terminating, decidability of type-checking is possible even in the presence of non-terminating programs at the type-level.

Our motivation to develop $\lambda_\star^\mu$ is to use it as a simpler alternative to existing core languages for languages such as Haskell. The paper shows how many of programming language features of Haskell, including some of the latest extensions, can be encoded in $\lambda_\star^\mu$ via a source language. In particular the source language supports *algebraic datatypes*, *higher-kinded types*, *nested datatypes*, *kind polymorphism* [] and *datatype promotion* []. This result is interesting because $\lambda_\star^\mu$ is a minimal calculus with only 8 language constructs and a single syntactic sort. In contrast the latest versions of System F_C (Haskell's core language) have multiple syntactic sorts and dozens of language constructs. Even if support for equality and coercions, which constitutes a significant part of System F_C, would be removed the resulting language would still be significantly larger and more complex than $\lambda_\star^\mu$.

BRUNO: $\lambda_\star^\mu$ sacrifices the convinience of use of type-level computation to gain the ability of doing arbitrary general recursion at the term level. We believe $\lambda_\star^\mu$ is particularly well-suited as a core for Haskell-like languages. In particular the treatment of type-level computation shares similar ideas with Haskell. Although Haskell's surface language provides a rich set of mechanisms to do type-level computation [], the core language lacks fundamental mechanisms todo type-level computation. In particular, like in $\lambda_\star^\mu$, type equality is purely syntactic (modulo alpha-conversion).

In summary, the contributions of this work are:

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

2. Overview

This section informally introduces the main features of $\lambda_\star^\mu$. In particular, this section shows how the explicit casts in $\lambda_\star^\mu$ can be used instead of the typical conversion rule present in calculi such as the calculus of constructions. The formal details of $\lambda_\star^\mu$ are presented in §4. JEREMY: to distinguish code from λC_{suf} and $\lambda_\star^\mu$, we may want to use different fonts, e.g., `typewriter font` for λC_{suf}

2.1 The Calculus of Constructions and the Conversion Rule

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

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

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

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

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

Decidability of Type-Checking and Strong Normalization While the conversion rule in λC brings a lot of convenience, an unfortunate consequence is that it couples decidability of type-checking with strong normalization of the calculus [8]. However strong normalization does not hold with general recursion. This is because due to the conversion rule, any non-terminating term would force the type checker to go into an infinitely loop (by constantly applying the conversion rule without termination), thus rendering the type system undecidable.

To illustrate the problem of the conversion rule with general recursion, let us consider a somewhat contrived example. Suppose that d is a "dependent type" that has type $\mathsf{Int} \to \star$. With general recursion at hand, we can image a term z that has type $d\,\mathsf{loop}$, where loop stands for any diverging computation of type Int. What would happen if we try to type check the following application:

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

Under the normal typing rules of λC, the type checker would get stuck as it tries to do β-equality on two terms: $d\,3$ and $d\,\mathsf{loop}$, where the latter is non-terminating. BRUNO: show simple example. Explain issue better. JEREMY: done!

2.2 An Alternative to the Conversion Rule: Explicit Casts

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

Beta Reduction The first of the two type conversions $\mathsf{cast}_\downarrow$, allows a type conversion provided that the resulting type is a *beta reduction* of the original type of the term. The use of $\mathsf{cast}_\downarrow$ is better explained by the following simple example. Suppose that

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

and term z has type

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

$g\,z$ is an ill-typed application, whereas $g\,(\mathsf{cast}_\downarrow\,z)$ is well-typed. This is witnessed by $(\lambda y : \star.y)\ \mathsf{Int} \to_\beta \mathsf{Int}$, which is a beta reduction of $(\lambda y : \star.y)\ \mathsf{Int}$. BRUNO: explain why this is a reduction JEREMY: done!

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. Let us revisit the example from §2.1. In $\lambda_\star^\mu$, $f\,3$ is an ill-typed application. Instead we must write the application as

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

BRUNO: how to put a space before 3? JEREMY: fixed! Intuitively, $\mathsf{cast}^\uparrow$ is doing a type conversion, as the type of 3 is Int, and

$(\lambda y : \star . y)\,\mathsf{Int}$ is the beta expansion of Int (witnessed by $(\lambda y : \star . y)\,\mathsf{Int} \to_\beta \mathsf{Int}$). BRUNO: explain why this is a beta expansion JEREMY: done! 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 are more than one choices for beta expansions (e.g., $1+2$ and $2+1$ are both the beta expansions of 3). BRUNO: explain why for beta expansions we need to provide the resulting type as argument JEREMY: done!

A final point to make is that the cast rules specify *one-step* reduction. This enables us to have more control over type-level computation. The full technical details about cast rules are presented in §4.

2.3 Decidability without Strong Normalization

With explicit type conversion rules the decidability of type-checking no longer depends on the normalization property. A nice consequence of this is that the type system remains decidable even in the presence of non-terminating programs at type level.

To illustrate, let us consider the same example discussed in §2.1. Now the type checker will not get stuck when type-checking the following application:

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

where the type of z is $d\,\mathsf{loop}$. This is because in $\lambda_\star^\mu$, the type checker only does syntactic comparison between $d\,3$ and $d\,\mathsf{loop}$, instead of β-equality. Therefore it rejects the above application as ill-typed. Indeed it is impossible to type-check the 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., $(\lambda x : d\,3 . x)(\mathsf{cast}_\downarrow(\mathsf{cast}_\downarrow \ldots z))$). But it is impossible to write such program in reality.

In summary, $\lambda_\star^\mu$ achieves the decidability of type checking by explicitly controlling type-level computation. which is independent of the normalization property, while supporting general recursion at the same time.

2.4 Recursion and Recursive Types

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

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

Recursion The μ primitive can be used to define recursive functions. For example, the factorial function:

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

The above recursive definition works because of the dynamic semantics of the μ primitive:

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

which is exactly doing recursive unfolding of itself.

It is worth noting that the type τ in S_MU is not restricted to function types. This extra freedom allows us to define a record of mutually recursive functions as the fixed point of a function on records.

Recursive types In the literature on type systems, there are two approaches to recursive types, namely *equi-recursive* and *iso-recursive*. The *iso-recursive* approach treats a recursive type and its unfolding as different, but isomorphic. The isomorphism between a recursive type and its one step unfolding is witnessed by traditionally fold and unfold operations. In $\lambda_\star^\mu$, the isomorphism is witnessed by first $\mathsf{cast}^\uparrow$, then $\mathsf{cast}_\downarrow$. BRUNO: Explain that the casts generalize fold and unfold! JEREMY: done! At first sight, the cast rules share some similarities with fold and unfold, but $\mathsf{cast}^\uparrow$ and $\mathsf{cast}_\downarrow$ actually generalize fold and unfold: they can convert any types, not just recursive types. To demonstrate the use of the cast rules, let us consider a classic example of a recursive type, the so-called "hungry" type [19]: $H = \mu\sigma : \star . \mathsf{Int} \to \sigma$. A term z of type H can accept any number of integers and return a new function that is hungry for more, as illustrated below:

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

Call-by-Name Due to the unification, the *call-by-value* evaluation strategy does not fit in our setting. In call-by-value evaluation, recursion can be expressed by the recursive binder μ as $\mu f : T \to T . E$ (note that the type of f is restricted to function types). Since we don't want to pose restrictions on the types, the *call-by-name* evaluation is a sensible choice. BRUNO: Probably needs to be improved. I'll came back to this later!

2.5 Logical Inconsistency

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

One consequence of adding general recursion to the type system is that the logical consistency of the system is broken. This is a deliberate design decision, since our goal is to model languages like Haskell, which are logically inconsistent as well.

In light of the fact that we decide to give up consistency, we take another step further by declaring that the kind $\star$ is of type $\star$. As it turns out, having this rule adds expressiveness and simplifies our system. We return to this issue in §7.

2.6 Encoding Datatypes

The explicit type conversion rules and the μ primitive facilitates the encoding of recursive datatypes and recursive functions over datatypes. While inductive datatypes can be encoded using either the Church or the Scott encoding, we adopt the Scott encoding as it encodes case analysis, making it more convenient to encode pattern matching. We demonstrate the encoding method using a simple datatype as a running example: Peano numbers.

The datatype declaration for Peano numbers in Haskell is:

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

In the Scott encoding, the encoding of the *Nat* datatype reflects how its two constructors are going to be used. Since *Nat* is a recursive datatype, we have to use recursive types at some point to reflect its recursive nature. As it turns out, the typed Scott encoding of *Nat* is:

$$\mu\ X : \star . \Pi\ B : \star . B \to (X \to B) \to B$$

The function type $B \to (X \to B) \to B$ demystifies the recursive nature of *Nat*: B corresponds to the type of the constructor *Z*, and $X \to B$ corresponds to the type of the constructor *S*. The intuition is that any recursive use of the datatype in the data constructors is replaced with the variable (X in the case) bound by μ, and we make the resulting variable (B in this case) universally quantified so that elements of the datatype with different result types can be used in the same program [15].

Its two constructors can be encoded correspondingly via the cast rules:

$$Z = \mathsf{cast}^{\uparrow}\ [Nat]\ (\lambda B : \star.\, \lambda z : B.\, \lambda f : Nat \to B.\, z)$$
$$S = \lambda n : Nat.\, \mathsf{cast}^{\uparrow}\ [Nat]\ (\lambda B : \star.\, \lambda z : B.\, \lambda f : Nat \to B.\, f\ n)$$

Thanks to the cast rules, we can make use of the $\mathsf{cast}^{\uparrow}$ operation to do type conversion between the recursive type and its unfolding.

The last example defines a recursive function that adds two natural numbers:

$$\mu\, f : Nat \to Nat \to Nat.\, \lambda n : Nat.\, \lambda m : Nat.\\ (\mathsf{cast}_{\downarrow}\ n)\ Nat\ m\ (\lambda n' : Nat.\, S\ (f\ n'\ m))$$

The above definition quite resembles case analysis commonly seen in modern functional programming languages. (We formalize the encoding of case analysis in §6.)

3. λC_{suf} by Example

BRUNO: Wrong title! This section is not about $\lambda^{\mu}_{\star}$; it is about source languages that can be built on top of name! JEREMY: this name for the moment

BRUNO: General comment is that, although the material is good, the text is a bit informally written. Text needs to be polsihed. Also the text is lacking references.

This sections shows a number of programs written in the surface language λC_{suf}, which in built on top of $\lambda^{\mu}_{\star}$. Most of these examples either require non-trivial extensions of Haskell, or are non-trivial to encode in dependently typed language like Coq or Agda. The formalization of the surface language is presented in §6.

Datatypes Conventional datatypes like natural numbers or polymorphic lists can be easily defined in λC_{suf}, BRUNO: This is not name; its the source language built on top of name! JEREMY: changed as in Haskell. For example, below is the definition of polymorphic lists:

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

Because λC_{suf}BRUNO: You'll have to stop referring to $\lambda^{\mu}_{\star}$ in this section. You may want to consider giving the source language a name. JEREMY: changed is explicitly typed, each type parameter needs to be accompanied with a corresponding kind expression. The use of the above datatype is best illustrated by the *length* function:

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

Higher-kinded Types Higher-kinded types are types that take other types and produce a new type. To support higher-kinded types, languages like Haskell have to extend their existing core languages to account for kind expressions. (The existing core language of Haskell, System FC, is an extension of System F_{ω} [12], which naively supports higher-kinded types.) BRUNO: Probably want to mention F_{ω} JEREMY: done! Given that λC_{suf} subsumes System F_{ω}, we can easily construct higher-kinded types. We show this by an example of encoding the *Functor* class:

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

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

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

HOAS *Higher-order abstract syntax* is a representation of abstract syntax where the function space of the meta-language is used to encode the binders of the object language. Because of the recursive occurrence of the datatype appears in a negative position (i.e., in the left side of a function arrow) BRUNO: explain where! JEREMY: done!, systems like Coq and Agda would reject such programs using HOAS due to the restrictiveness of their termination checkers. However λC_{suf} is able to express HOAS in a straightforward way. We show an example of encoding a simple lambda calculus:

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

Next we define the evaluator for our lambda calculus. As noted by [11], the evaluation function needs an extra function *reify* to invert the result of evaluation.

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

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

Evaluation of a lambda expression proceeds as follows:

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

Fix as a Datatype The type-level *Fix* is a good example to demonstrate the expressiveness of λC_{suf}. The definition is:

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

The record notation also introduces the selector function: $out : (f : \star \to \star) \to Fix\ f \to f\ (Fix\ f)$. The *Fix* datatype is interesting in that now we can define recursive datatypes in a non-recursive way. For instance, a non-recursive definition for natural numbers is:

$$\textbf{data}\ NatF\ (self : \star) = Zero \mid Succ\ self;$$

And the recursive version is just a synonym:

$$\textbf{let}\ Nat : \star = Fix\ NatF$$

Note that now we can use the above *Nat* anywhere, including the left-hand side of a function arrow, which is a potential source of non-termination. The termination checker of Coq or Agda is so conservative that it would brutally reject the definition of *Fix* to avoid the above situation. BRUNO: show example? JEREMY: done! However in λC_{suf}, where type-level computation is explicitly controlled, we can safely use *Fix* in the program.

Given *fmap*, many recursive shcemes can be defined, for example we can have *catamorphism* [16] BRUNO: reference? JEREMY: done! or generic function fold:

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

Kind Polymophism In Haskell, System FC [29] BRUNO: reference JEREMY: done! was proposed to support kind polymorphism. However it separates expressions into terms, types and kinds, which complicates both the implementation and future extensions. λC_{suf} natively allows datatype definitions to have polymorphic kinds. Here is an example, taken from [29], of a datatype that benefits from kind polymophism: a higher-kinded fixpoint combinator:

$$\begin{aligned}
&\textbf{data } \mathit{Mu}\ (k : \star)\ (f : (k \to \star) \to k \to \star)\ (a : k) = \\
&\quad \mathit{Roll}\ (f\ (\mathit{Mu}\ k\ f)\ a);
\end{aligned}$$

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

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

Nested Datatypes A nested datatype [5] BRUNO: reference JEREMY: done!, also known as a *non-regular* datatype, is a parametrised datatype whose definition contains different instances of the type parameters. Functions over nested datatypes usually involve polymorphic recursion. We show that λC_{suf} is capable of defining all useful functions over a nested datatype. A simple example would be the type *Pow* of power trees, whose size is exactly a power of two, declared as follows:

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

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

$$\begin{aligned}
&\textbf{let } \mathit{Pow} : \star \to \star = \mathsf{mu}\ X : \star \to \star. \\
&\quad \lambda a : \star.\, (B : \star) \to (a \to B) \to (X\ (\mathit{PairT}\ a) \to B) \to B
\end{aligned}$$

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

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

Data Promotion BRUNO: what is the point that we are trying to make with this example? Title is wrong; should be about the point, not about the particular example! JEREMY: This section shows we can do data promotion much more easily than in Haskell

Haskell needs sophisticated extensions [29] in order for being able to use ordinary datatypes as kinds, and data constructors as types. With the full power of dependent types, data promotion is made trivial in λC_{suf}.

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:

$$\begin{aligned}
&\textbf{data } \mathit{PTree}\ (n : \mathit{Nat}) = \mathit{Empty} \\
&\quad \mid \mathit{Fork}\ \mathit{nat}\ (\mathit{PTree}\ (S\ n))\ (\mathit{PTree}\ (S\ n));
\end{aligned}$$

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

$$\mathit{Fork}\ Z\ 1\ (\mathit{Empty}\ (S\ Z))\ (\mathit{Empty}\ (S\ Z))$$

If we accidentally write the wrong depth, for example:

$$\mathit{Fork}\ Z\ 1\ (\mathit{Empty}\ (S\ Z))\ (\mathit{Empty}\ Z)$$

The above will fail to pass type checking.

BRUNO: Two questions: firstly does it work? secondly do we support GADT syntax now? JEREMY: changed to a simple binary tree example

BRUNO: More examples? closed type families; dependent types? JEREMY: had hard time thinking of a simple, non-recursive example for type families

4. A Dependently Typed Calculus with Casts

In this section, we present the $\lambda_\star$ calculus. This calculus is very close to the calculus of contructions, except for two 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. Like the calculus of constructions, $\lambda_\star$ has decidable type-checking. However, unlike λC the proof of decidability of type-checking does not require the strong normalization of the calculus. In rest of this section, we demonstrate the syntax, operational semantics, typing rules and meta-theory of $\lambda_\star$.

type-level

4.1 Syntax

Figure 1 shows the syntax of $\lambda_\star$, including expressions, contexts and values. $\lambda_\star$ uses a unified syntactic representation for different levels of expressions by following the *pure type system* (PTS) representation of λC. Therefore, there is no syntactic distinction between terms, types or kinds. This design brings the economy for type checking, since one set of rules can cover all syntactic levels. By convention, we use metavariables τ and σ for an expression on the type-level position and e for one on the term level.

Type of Types Traditionally 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. In $\lambda_\star$, 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 implicit conversion rule of λC with *one-step* explicit type conversion. They represent two directions of type conversion: $\mathsf{cast}_{\downarrow}$ stands for the β-reduction of types, while $\mathsf{cast}^{\uparrow}$ is the inverse (Examples will be given in later typing sections).

Though cast primitives make the syntax verbose when type conversion is heavily used, the implementation of type checking is simplified because typing rules of $\lambda_\star$ become type-directed without λC's implicit conversion rule. Considering the core language is compiler-oriented, end-users will not directly use them. Some type

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

Figure 1. Syntax of $\lambda_\star$

conversions can be generated through the translation of the source language (§6).

4.2 Operational Semantics

Figure 2 shows the *call-by-name* operational semantics, defined by one-step reduction. Two base cases include S_BETA for β-reduction and S_CASTDOWNUP for cast canceling. Two inductive case, S_APP and S_CASTDOWN, define reduction in the head position of an application, and in the $\mathsf{cast}_{\downarrow}$ inner expression respectively. The reduction rules are *weak* in the sense that it is not allowed to reduce inside a λ-term or $\mathsf{cast}^{\uparrow}$-term which is viewed as a value (see Figure 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, which is possible to be infinite. The multi-step reduction can be defined as follows:

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

For a consecutive sequence of reductions with n steps, we use the notation $\longrightarrow_n$ to denote the relation between the initial and final expressions:

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

4.3 Typing

Figure 3 gives the *syntax-directed* typing rules of $\lambda_\star$, including rules of context well-formedness $\vdash \Gamma$ and expression typing $\Gamma \vdash e : \tau$. Note that there is only a single set of rules for expression typing, because there is no distinction of different syntactic levels.

Most typing rules are quite standard. We write $\vdash \Gamma$ if a context Γ is well-formed. Note that there is only a single sort $\star$, we use $\Gamma \vdash \tau : \star$ to check if τ is a well-formed type. Rule T_AX is the 'type-in-type' axiom. Rule T_VAR checks the type of variable x from the valid context. Rules T_APP and T_LAM check the validity of application and abstraction. Rules T_PI check the type well-formedness of the dependent function.

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 (see §2.1). The relation between the original

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

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

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

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

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

Figure 2. Operational semantics of $\lambda_\star$

and converted type is defined by one-step reduction (see 2). Specifically speaking, if given a judgement $\Gamma \vdash e : \tau_2$ and relation $\tau_1 \longrightarrow \tau_2 \longrightarrow \tau_3$, then $\mathsf{cast}^{\uparrow}\, [\tau_1]\, e$ expands the type of e from τ_2 to τ_1, while $\mathsf{cast}_{\downarrow}\, e$ reduces the type of e from τ_2 to τ_3. For example, assume $\Gamma \vdash e_1 : \mathsf{Int}$ and $\Gamma \vdash e_2 : (\lambda x : \star . x)\ \mathsf{Int}$. Note that the following reduction holds:

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

Thus, we can obtain the following derivation of e_1 and e_2:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathsf{Int} \\ \Gamma \vdash (\lambda x : \star . x)\ \mathsf{Int} : \star \qquad (\lambda x : \star . x)\ \mathsf{Int} \longrightarrow \mathsf{Int}\end{array}}{\Gamma \vdash (\mathsf{cast}^{\uparrow}\, [(\lambda x : \star . x)\ \mathsf{Int}]\ e_1) : (\lambda x : \star . x)\ \mathsf{Int}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_2 : (\lambda x : \star . x)\ \mathsf{Int} \\ \Gamma \vdash \mathsf{Int} : \star \qquad (\lambda x : \star . x)\ \mathsf{Int} \longrightarrow \mathsf{Int}\end{array}}{\Gamma \vdash (\mathsf{cast}_{\downarrow}\, e_2) : \mathsf{Int}}$$

BRUNO: More details please! Show worked out examples with a typing derivation. LINUS: Fixed.

Importantly, in $\lambda_\star$ 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.

Type in Type BRUNO: I have moved this text from the previous section. This is where you should talk about the *typing* consequences of the * : * axiom. LINUS: Noted with thanks. In the context of λC, if a term x has the type τ_1, and τ_2 is a type, i.e. $x : \tau_1 : \star$ and $\tau_2 : \square$, we call the type $\Pi\, x : \tau_1 . \tau_2$ a *dependent product*. $\lambda_\star$ follows λC to use the same Π-notation to represent dependent function types.

However, a higher-kind polymorphic function type such as $\Pi\, x : \square . x \rightarrow x$ is not allowed in λC, because $\square$ is the highest sort that can not be typed. While Π-notation in $\lambda_\star$ is more expressive and does not have such limitation because of the axiom $\star : \star$. In the surface language, we interchangeably use the arrow form $(x : \tau_1) \rightarrow \tau_2$ of the product for clarity. By convention, we also use the syntactic sugar $\tau_1 \longrightarrow \tau_2$ to represent the product if x does not occur free in τ_2.

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

$\boxed{\vdash \Gamma}$ 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

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

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

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

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

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

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

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

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

4.4 Meta-theory

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

Decidability of Type Checking For the decidability, we need to show there exists a type checking algorithm, which never loops forever and returns a unique type for a well-formed expression e. This is done by induction on the length of e and ranging over typing rules. Most expression typing rules, which have only typing judgements in premises, are already decidable by induction hypothesis. Thus, it is straightforward to follow the syntax-directed judgement to derive a unique type checking result.

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

Lemma 4.3 (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 no such e'.*

Proof. By induction on the structure of e. □

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

Theorem 4.4 (Decidability of Type Checking $\lambda_\star$)**.** *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. □

Note that when proving the decidability of type checking, we do not rely on the normalizing property. Because explicit type conversion rules use one-step reduction, which already has a decidable checking algorithm according to Lemma 4.3. We do not need to further require the normalization of terms. This is different from the proof for λC which requires the language is normalizing [14]. Because λC's conversion rule needs to examine the β-equivalence of terms, which is decidable only if every term has a normal form.

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

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

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

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

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

Lemma 4.5 (Decidability of n-step Reduction)**.** *The n-step reduction $\longrightarrow_n$ is called decidable if given e there is a unique e' such that $e \longrightarrow_n e'$ or no such e'.*

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

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

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

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

Lemma 4.6 (Substitution)**.** *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 $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$. □

Theorem 4.7 (Subject Reduction)**.** *If $\Gamma \vdash e : \sigma$ and $e \twoheadrightarrow e'$ then $\Gamma \vdash e' : \sigma$.*

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

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

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

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

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

Figure 4. Syntax and semantics changes for general recursion

Proof. (*Sketch*) We prove the case for one-step reduction, i.e. $e \longrightarrow e'$. The lemma can follow by induction on the number of one-step reductions of $e \twoheadrightarrow e'$. The proof is by induction with respect to the definition of one-step reduction $\longrightarrow$. □

Theorem 4.8 (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 $\varnothing \vdash e : \sigma$. □

5. Dependent Types with General Recursion

In this section we present $\lambda_\star^\mu$: an extension of $\lambda_\star$ with a general recursion contruct. The general recursion is polymorphic and has a uniform representation on both term level and type level. The same construct works both as a term-level fixpoint and recursive type. The addition of general recursion does not break decidability of type-checking nor type-safety.

5.1 Syntax and Semantics

Figure 4 shows the changes of extending $\lambda_\star$ to $\lambda_\star^\mu$ with general recursion. The changes are subtle, since we add only one more primitive, reduction rule and typing rule for general recursion. Nevertheless general recursion allows a large number of programs that can be expressed in programming languages such as Haskell to be expressed in $\lambda_\star^\mu$ as well.

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

5.2 Recursion as Term and Type

Term-level Recursion In $\lambda_\star^\mu$, 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 in e with the whole μ-term itself, resulting in the unrolling $e[x \mapsto \mu\, x : \tau . e]$. The μ-term 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 μ-term is not treated as a value. This is different from conventional term-level fixpoint that is usually treated as values [7].

Recall the factorial function example (§**??**), which can be defined as a μ-term as follows:

$$
\begin{array}{l}
\mathsf{fact} \triangleq \mu f : \mathsf{Int} \to \mathsf{Int}. \\
\qquad \lambda x : \mathsf{Int}.\ \mathbf{if}\ x \equiv 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (f\ (x - 1))
\end{array}
$$

By rule T_Mu, the type of fact is Int $\to$ Int. Thus, we can apply fact to an integer, say 1, and will get an integer as the result. By rules S_Mu and S_App, we can evaluate the term fact 1 as follows:

$$
\begin{array}{l}
\quad \mathsf{fact}\ 1 \\
\longrightarrow (\lambda x : \mathsf{Int}.\ \mathbf{if}\ x \equiv 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (\mathsf{fact}\ (x - 1)))\ 1 \\
\longrightarrow \mathbf{if}\ 1 \equiv 0\ \mathbf{then}\ 1\ \mathbf{else}\ 1 \times (\mathsf{fact}\ (1 - 1)) \\
\longrightarrow 1 \times (\mathsf{fact}\ (1 - 1)) \\
\longrightarrow 1 \times (\mathsf{fact}\ 0) \longrightarrow\ \dots \longrightarrow\ 1 \times 1 \longrightarrow\ 1.
\end{array}
$$

Note that we never check if a μ-term can terminate or not, which is an undecidable halting 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. Later we will see type checking $\lambda_\star^\mu$ is still decidable. BRUNO: Show an example of the execution, maybe for fact(2) LINUS: Done.

Type-level Recursion On the type level, $\mu\, x : \tau . e$ works as a *iso-recursive* type, a kind of recursive type that is not equal but only isomorphic to its unrolling. Normally, we need to add two more primitives fold and unfold for the iso-recursive type to map back and forth between the original and unrolled form:

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

where the operators satisfy the following typing rules:

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

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

BRUNO: Show the rules for fold and unfold here to help you make the argument. LINUS: Figure added to show the relationship.

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 and have the same functionality. Assume there exist expressions e_1 and e_2 such that

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

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

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

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

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

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

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

5.3 Decidability of Type Checking and Type Safety

BRUNO: I think it is worth restating the lemmas here to aid the discussion. No need to show all lemmas, just the main ones: decidability of type-checking and type-safety. LINUS: Revised. We give the proof of the decidability of type checking $\lambda_\star$ in §4.4 without requiring normalization. The cast rules are critical for decidability, which rely on checking if one type can be reduced to another in one step. When we introduce general recursion into the language, if we can make sure the newly added and original typing rules are still decidable, the decidability of type checking will still follow in $\lambda_\star^\mu$.

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

Theorem 5.1 (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.*

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

Theorem 5.2 (Subject Reduction)**.** *If $\Gamma \vdash e : \sigma$ and $e \twoheadrightarrow e'$ then $\Gamma \vdash e' : \sigma$.*

Theorem 5.3 (Progress)**.** *If $\varnothing \vdash e : \sigma$ then either e is a value v or there exists e' such that $e \longrightarrow e'$.*

6. Surface language

In this section, we present a surface language λC_{suf}, built on top of $\lambda_\star^\mu$ with features that are convenient for functional programming: user-defined datatypes, and pattern matching. 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 extended 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.

6.1 Extended Syntax

The full syntax of λC_{suf} is defined in Figure 5. Compared with $\lambda_\star^\mu$, λC_{suf} has a new syntax category: a program, consisting of a list of datatype declarations, followed by a expression. An *algebraic data type D* is introduced as a top-level **data** declaration with its *data constructors*. For the purpose of presentation, we sometimes adopt the following syntactic convention:

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

The type of a data constructor K has the form:

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

BRUNO: this looks a bit odd for a number of reasons: firstly why to insist on having the quantified variables in the same order as the arguments in the constructor? Secondly it seems that all other arguments cannot be dependently typed? It seems to me that

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

where all variables u are bound ($u \in \overline{x}$) would be better. JEREMY: changed! $(\,\overline{u : \kappa}\,)$ are for the arguments of a type constructor, $(\,\overline{x : T}\,)$ are for the arguments of a data constrcutor Note that the use of the dependent product in the type of a data constructor (e.g., $(\,\overline{u : \kappa}\,)$) makes it possible to let the type of some type constructor arguments depend on other type constructor arguments, while in Haskell, this is not possible, becuase the arrow $\to$ can be seen as an independent product type. The **case** expression is conventional, used to break up values built with data constructors. The patterns of a case expression are flat (no nested patterns), and bind value variables.

For the sake of programming, λC_{suf} employs some syntactic sugar. A non-dependent function type can be written as $T_1 \to T_2$. A dependent function type $\Pi\, x : T_1.T_2$ is abbreviated as $(x : T_1) \to T_2$ for easy reading. We also introduce a Haskell-like record syntax, which is desugared to datatypes with accompanying selector functions.

6.2 Extended Typing Rules

BRUNO: For typing and translation show only one figure (Figure 8), since the typing figure is just a subset. We can use gray to highlight the parts which belong to the translation. JEREMY: adjusted!

Figure 6 defines the type system of the surface language (ignore the gray parts for the moment). Several new typing judgments appear in the type system. The use of different subscripts of the judgments is to be distinct from the one used in $\lambda_\star^\mu$. Most rules of the type system 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). Two judgments $\Sigma \vdash_{\mathsf{pg}} pgm : T$ and $\Sigma \vdash_{\mathsf{d}} decl : \Sigma'$ are of the essence to the type checking of the surface language. The former type checks a whole program, and the latter type checks datatype declarations.

Rule TRPGM_PGM type checks a whole program. It first typechecks the declarations, which in return gives a new typing environment. Combined with the original environment, it then continues to type check the expression and 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 application (of kind $\star$). Note that since our system adopts $\star : \star$, this means we can express kind polymophism for datatypes. Finally it make sure the types of data constructors are valid.

Rules TR_CASE and TRPAT_ALT handle the type checking of case expressions. The conclusion of TS_CASE binds the right types to the scrutinee E_1 and alternatives $\overline{p \Rightarrow E_2}$. The first premise of TPAT_ALT binds the actual type constructor arguments to $\overline{u}$. The second premise checks whether the types of the right-hand sides of each alternative, instantiated to the actual type constructor arguments $\overline{u}$, are equal. Finally the third premise checks the well-formedness of the types of data constructor arguments. BRUNO: Mention that we do not support refinement, as in GADTs? JEREMY: done!

As can be seen, currently λC_{suf} does not support refinement on the final result of each data constructor, as in GADTs. However, our encoding method does support some form of GADTs, as is discussed in §**??**.

6.3 Translation Overview

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

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

It states that $\lambda_\star^\mu$ expression e is the translation of the surface expression E of type T. The gray parts in Figure 6 defines the translation rules. BRUNO: Any partiular reasons for this? JEREMY: deleted

pgm	$::=$		Program
	$\mid$	$\overline{decl}\,; E$	Declarations
$decl$	$::=$		Declarations
	$\mid$	$\mathbf{data}\, D\, \overline{u:\kappa} \;=\; \overline{K\,\overline{T}}$	Datatype declaration
$u,\ \upsilon$	$::=$		Atoms
	$\mid$	x	Variable
	$\mid$	K	Data constructor
p	$::=$		Patterns
	$\mid$	$K\,\overline{x:T}$	Pattern
$E,\ T,\ S,\ \kappa$	$::=$		Expressions
	$\mid$	u	Atom
	$\mid$	$\star$	Type of types
	$\mid$	$E_1\ E_2$	Application
	$\mid$	$\lambda x : T.E$	Abstraction
	$\mid$	$\Pi\, x : T.S$	Product
	$\mid$	$\mathsf{cast}^{\uparrow}\,[T]\,E$	Cast up by expansion
	$\mid$	$\mathsf{cast}_{\downarrow}\,E$	Cast down by reduction
	$\mid$	$\mu\, x : T.E$	General recursion
	$\mid$	$\mathbf{case}\ E_1\ \mathbf{of}\ \overline{p \Rightarrow E_2}$	Case analysis
v	$::=$		Values
	$\mid$	$\star$	Type of types
	$\mid$	$\lambda x : T.E$	Abstraction
	$\mid$	$\Pi\, x : T.S$	Product
	$\mid$	$\mathsf{cast}^{\uparrow}\,[T]\,E$	Cast up by expansion
	$\mid$	$K\,\overline{u}\,\overline{E}$	Data constructor
Σ	$::=$		Contexts
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Sigma, u : T$	Atom binding

				Syntactic Sugar
$T_1 \to T_2$	$\triangleq$	$\Pi\, x : T_1.T_2$	$x \notin \mathsf{FV}(T_2)$	Function type
$(x : T_1) \to T_2$	$\triangleq$	$\Pi\, x : T_1.T_2$	$x \in \mathsf{FV}(T_2)$	Dependent function type
$\mathsf{cast}^{n}_{\uparrow}\,[T_1]\,E$	$\triangleq$	$\mathsf{cast}^{\uparrow}[T_1](\mathsf{cast}^{\uparrow}[T_2](\ldots(\mathsf{cast}^{\uparrow}\,[T_n]\,E)\ldots))$		Cast up in n steps
$\mathsf{cast}^{n}_{\downarrow}\,E$	$\triangleq$	$\underbrace{\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow}(\ldots(\mathsf{cast}_{\downarrow}}_{n}\,E)\ldots))$		Cast down in n steps
$\mathbf{data}\, R\,\overline{u:\kappa} \;=\; K\{\,\overline{N:T}\,\}$	$\triangleq$	$\mathbf{data}\, R\,\overline{u:\kappa} \;=\; K\,\overline{T}\,;$ $\mathbf{let}\ N_i : (\,\overline{u:\kappa}\,) \to R\,\overline{u} \;\to T_i =$ $\lambda\,\overline{u:\kappa}\,.\lambda y : R\ \overline{u}\,.\mathbf{case}\ y\ \mathbf{of}\ K\ \overline{x:T}\ \Rightarrow x_i\ \mathbf{in}$	$\forall i$	Record

Figure 5. Syntax of the surface language

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 expression, casting it down to the right type. It is then applied to the result type of the body expression and a list of translated $\lambda^{\mu}_{\star}$ expressions of its alternatives. Rule TRPAT_ALT tells how to translate each alternative. Basically it translates an alternative into a lambda abstraction, where each bound variable in the pattern corresponds to a bound variable 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. Note that due to the rigidness of the translation, pattern matching must be exhaustive, and the order of patterns matters (the same order as in the datatype declaration).

Rule TRDECL_DATA does the most heavy work and deserves further explanation. First of all, it results in an incomplete expression (as can be seen by the incomplete let expressions), The result expression is supposed to be prepended to the translation of the last expression to form a complete $\lambda^{\mu}_{\star}$ 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 mention of the datatype in the data constructor parameters is replaced with the recursive variable X. Note that for the moment, the result type variable α is restricted to have kind $\star$. This could pose difficulties when translating GADTs as we will discussion in the 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.

6.4 Type-safefy of Translation

JEREMY: put Linus's theorem here

$\boxed{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma}$ Context translation

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

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

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

$$\frac{\Sigma \vdash_{\mathsf{s}} (\,\overline{u : \kappa}\,) \to \star : \star \rightsquigarrow (\,\overline{u : \sigma}\,) \to \star \qquad \overline{\Sigma, D : (\,\overline{u : \kappa}\,) \to \star,\ \overline{u : \kappa} \vdash_{\mathsf{s}} \overline{T} \to D\,\overline{u} : \star \rightsquigarrow \overline{\tau} \to D\,\overline{u}}}{\Sigma \vdash_{\mathsf{d}} (\mathbf{data}\, D\,\overline{u : \kappa} = \overline{K\,\overline{T}}\,) : (D : (\,\overline{u : \kappa}\,) \to \star,\ \overline{K : (\,\overline{u : \kappa}\,) \to \overline{T} \to D\,\overline{u}}\,) \rightsquigarrow e}\ \textsc{TRdecl\_Data}$$

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

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

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

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

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

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

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

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

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

$$\frac{\Sigma \vdash_{\mathsf{s}} E : T_2 \rightsquigarrow e \qquad \Sigma \vdash_{\mathsf{s}} T_1 : \star \rightsquigarrow \tau_1 \qquad T_1 \longrightarrow T_2}{\Sigma \vdash_{\mathsf{s}} (\mathsf{cast}^{\uparrow}\,[T_1]\,E) : T_1 \rightsquigarrow \mathsf{cast}^{\uparrow}\,[\tau_1]\,e}\ \textsc{TR\_CastUp}$$

$$\frac{\Sigma \vdash_{\mathsf{s}} E : T_1 \rightsquigarrow e \qquad \Sigma \vdash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2 \qquad T_1 \longrightarrow T_2}{\Sigma \vdash_{\mathsf{s}} (\mathsf{cast}_{\downarrow}\,E) : T_2 \rightsquigarrow \mathsf{cast}_{\downarrow}\,e}\ \textsc{TR\_CastDown}$$

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

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

Figure 6. Type directed translation rules of the surface language

7. Related Work

There is a lot work on bring full-spectrum dependent types to the practical programming world.

Unification of Terms, Types, and Kinds BRUNO: This subsection should start by talking about pure type systems and then move on to the work of Henk and others. JEREMY: done!

Pure Type Systems [4] show how a while family of type systems can be implemented using just a single syntactic form. This line of our work is largely inspired by Henk [13], where they are the first to use the so-called *lambda cube* as a typed intermediate language, unifying all three levels. Since the implicit conversion of the lambda cube is not syntax-directed, they come up with a approach to strategically distribute the conversion rule over the other typing rules. In retrospect, Henk is quite conservative in terms of type-level computation. Actually it is not even a dependently typed language, as they clearly state that they don't allow types to depend on terms. As for recursion, even though it has a full lambda calculus at the type level, recursion is disallowed.

Another recent work on dependently typed language based on the same syntactic category is Zombie [7, 23], where terms, types and the single kind $\star$ all reside in the same level. The language is based on a call-by-value variant of lambda calculus. One beautiful thing about Zombie is that it is composed of two fragments: a logical fragment and a programmatic fragment, so that it supports both partial and total programming. Even though Zombie has one syntactic category, it is still fairly complicated, as it tries to be both consistent as a logic and pragmatic as a programming language.

$\Pi\Sigma$ [2] is another recently proposed dependently typed core language that resembles $\lambda_\star^\mu$, as there is no syntactic difference between terms and types.BRUNO:

General Recursion and Managed Type-level Computation As discussed in §5 BRUNO: where? words like before and after are a bad smell. In academic writing we use references. Just add the reference to the section. JEREMY: fixed, bringing general recursion blindly into the dependently typed world causes more trouble than convenience. There are many dependently typed languages that allow general recursion. Zombie approaches general recursion by separating a consistent sub-language, in which all expressions are known to terminate, from a programmatic language that supports general recursion. What is interesting about Zombie is that those two seemingly conflicting worlds can interact with each other nicely, without compromising the consistency property. The key idea of this is to distinguish between these two fragments by using a *consistency classifier* θ. When θ is L, it means the logical part, and P the program part. Like $\lambda_\star^\mu$, Zombie uses roll and unroll for iso-recursive types. To ensure normalization (in order for decidable type checking), it forbids the use of unroll in P, where the potential non-termination could arise.

$F^\star$ [26] also supports writing general-purpose programs with effects (e.g., state, exceptions, non-terminating, etc.) while maintaining a consistent core language. Unlike $\lambda_\star^\mu$, it has several sub-languages – for terms, proofs and so on. The interesting part of $F^\star$ lies in its kind system, which tracks the sub-languages and controls the interactions between them. The idea is to restrict the use of recursion in specifications and proofs while allowing arbitrary recursion in the program. They use $\star$ to denote program terms that may be effectful and divergent, and P for proofs that identify pure and total functions. In this way, they are able to ensure that fragments in a program used for building proof terms are never mixed with those that are potentially divergent. One difference from $\lambda_\star^\mu$ is that, types in $F^\star$ can only contain values but no non-value expressions, leading to its less expressiveness than $\lambda_\star^\mu$.

$\Pi\Sigma$ has a general mechanism for recursion. Like $\lambda_\star^\mu$, it uses one recursion mechanism for the definition of both types and programs. The key idea relies on lifted types and boxes: definitions are not unfolded inside boxes. The way they achieve decidable type checking is to use boxing to stop the infinite unfolding of the recursive call, at the cost of additional annotations stating where to lift, box and force. One concern of $\Pi\Sigma$ is that its metatheory is not yet formally developed.

Type in Type We are not the the first to embrace $\star : \star$ in the system. It has been long known that systems with $\star : \star$ (usually called system $\lambda *$) is inconsistent as a logic, in the sense that all types are inhabited. In this system, we can encode a variant of Russel's paradox, known as Girard's paradox [9].

The core language of the Glasgow Haskell Compiler, System FC [25] has been extended with type promotion [29] and kind equality [28]. The latter one introduces a limited form of dependent types into the system[1], which mixes up types and kinds. This causes no trouble for FC, since all kinds are already inhabited without the above extensions. $\Pi\Sigma$ has a impredicative universe of types with $\mathsf{Type} : \mathsf{Type}$ due to the support of general recursion. The surface language of Zombie also has the rule $\Gamma \vdash \mathsf{Type} : \mathsf{Type}$ [24].

The $\star : \star$ axiom makes it convenient to support kind polymorphism, among other language features. One concern is that it often causes type checking to be undecidable, if not dealt with carefully, as it allows to express divergent terms. However, as we explained in §4, this is not the case for $\lambda_\star^\mu$. Type checking in $\lambda_\star^\mu$ is decidable – all type-level computation is driven by finite cast operations, thus no potentially infinite reductions can happen in reality.

Encoding of Datatypes One thing $\lambda_\star^\mu$ differs from other functional programming languages is that all the high-level features in the surface language like datatypes, pattern matching and so on can be easily encoded into the core language. There is much work on encoding datatypes into various high-level languages. The classic Church encoding of datatypes into System F is detailed in the work of Bohm and Beraducci [6]. The Church encoding excels in implementing iterative or fold-like functions over algebraic datatypes, but is awkward in expressing general recursion, usually in a complex and insufficient way. An alternative encoding of datatypes is the so called *Scott* encoding. However Scott encoding is not typable in System F, as it needs recursive types to represent recursive datatypes. $\lambda_\star^\mu$ has all it needs to represent polymorphic and recursive datatypes. The explicit cast rules also makes it possible to encode GADTs, as can be seen in the last examples in §3. Currently we are investigating how the encoding of GADTs interact with the other language constructs. We leave this as future work.

Another line of related work is the *inductive defined types* in the Calculus of Inductive Constructions (CIC) [18], which is the underlying formal language of Coq. In CIC, inductive defined types can be represented by closed types in λC, so are the primitive recursive functionals over elements of the type. The limitation of their work is that functions over inductive defined types are definable only by primitive recursion, not general recursion. Conor McBride's work on *Observational Type Theory* (OTT) [1] shows the encoding of datatypes via $\mathcal{W}$-types.

BRUNO: work by Conor Mcbride on encoding datatypes? That needs to be discussed here. Also the calculus of inductive constructions should deserve some mention. JEREMY: added CIC, which work of Conor Mcbride? Towards Observational Type Theory is the paper i searched

[1] Richard A. Eisenberg is going to implement kind equality [28] into GHC. The implementation is proposed at `https://phabricator.haskell.org/D808` and related paper is at `http://www.cis.upenn.edu/~eir/papers/2015/equalities/equalities-extended.pdf`.

8. Discussion

More Type-level Computation

Erasure of* cast *Operators Explicit type cast operators $\mathsf{cast}^{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ are part of the core design of the language. They are generalized from **fold** and **unfold** of iso-recursive types. By convention, $\mathsf{cast}^{\uparrow}$ follows **fold** as a value in the language, which means it cannot be further reduced during the evaluation. Thus, it is likely to have many $\mathsf{cast}^{\uparrow}$ constructs leaving in terms after evaluation. This will be a performance issue when considering code generation – too many $\mathsf{cast}^{\uparrow}$ constructs will dramatically increase the size of the program, which causes runtime overhead and affects the performance of the language.

In fact, we can safely erase cast operators during code generation, because they do not perform any real transformation of a term other than changing its type.

Encoding of GADTs Our translation rules also open opportunity for encoding GADTs. In our experiment, we have several running examples of encoding GADTs. Below we show a GATD-encoded representation of well-scoped lambda terms using de Bruijn notation.

In this notation, a variable is represented as a number – its de Bruijn index, where the number k stands for the variable bound by the k's enclosing λ. Using the GADT syntax, below is the definition of lambda terms:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Fin} : \mathit{Nat} \to \star = \\
\quad\quad \mathit{fzero} : (n : \mathit{Nat}) \to \mathit{Fin}\ (S\ n) \\
\quad\ \mid \mathit{fsucc} : (n : \mathit{Nat}) \to \mathit{Fin}\ n \to \mathit{Fin}\ (S\ n); \\
\mathbf{data}\ \mathit{Term} : \mathit{Nat} \to \star = \\
\quad\quad \mathit{Var} : (n : \mathit{Nat}) \to \mathit{Fin}\ n \to \mathit{Term}\ n \\
\quad\ \mid \mathit{Lam} : (n : \mathit{Nat}) \to \mathit{Term}\ (S\ n) \to \mathit{Term}\ n \\
\quad\ \mid \mathit{App} : (n : \mathit{Nat}) \to \mathit{Term}\ n \to \mathit{Term}\ n \to \mathit{Term}\ n;
\end{array}
$$

The datatype *Fin n* is used to restrict the the de Brujin index, so that it lies between 0 to $n - 1$. The type of a closed term is simply $\mathit{Term}\ Z$, for instance, a lambda term $\lambda x.\,\lambda y.\,x$ is represented as (for the presentation, we use decimal numbers instead of Peano numbers):

$$\mathit{Lam}\ 0\ (\mathit{Lam}\ 1\ (\mathit{Var}\ 2\ (\mathit{fsucc}\ 1\ (\mathit{fzero}\ 0))))$$

If we accidentally write the wrong index, the program would fail to pass type checking.

We do not have space to present a complete encoding, but instead show the encoding of *Fin*:

$$
\begin{array}{l}
\mathbf{let}\ \mathit{Fin} : \mathit{Nat} \to \star = \\
\quad \mu\ X : \mathit{Nat} \to \star.\, \lambda a : \mathit{Nat}. \\
\quad\quad (B : \mathit{Nat} \to \star) \to \\
\quad\quad ((n : \mathit{Nat}) \to B\ (S\ n)) \to \\
\quad\quad ((n : \mathit{Nat}) \to X\ n \to B\ (S\ n)) \to \\
\quad\quad B\ a
\end{array}
$$

The key issue in encoding GATDs lies in type of variable B. In ordinary datatype encoding, B is fixed to have type $\star$, while in GADTs, its type is the same as the variable X (possibly higher-kinded). Currently, we have to manually interpret the type according to the particular use of some GADTs. We are investigating if there exits a general way to do that.

Using Recursive Functions at the Type Level

9. Conclusion

Conclusion and related work.

Acknowledgments

Thanks to Blah. This work is supported by Blah.

References

[1] T. Altenkirch and C. McBride. Towards Observational Type Theory. *Manuscript available online*, 2006. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.4451&rep=rep1&type=pdf`.

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

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

[4] H. Barendregt and H. Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1:124, 1991. . URL `http://dare.ubn.kun.nl/handle/2066/17240`.

[5] R. Bird and L. Meertens. Nested datatypes. *Mathematics of program construction*, 1422:52–67, 1998. . URL `http://www.springerlink.com/index/126u6246254u5263.pdf`.

[6] C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras, 1985. ISSN 03043975.

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

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

[9] T. Coquand. An analysis of girard's paradox. 1986.

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

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

[12] J.-Y. Girard. Interprétation fonctionelle et élimination des coupures de larithmétique dordre supérieur. 1972.

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

[14] L. Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993.

[15] Y. Mandelbaum and a. Stump. GADTs for the OCaml masses. *Workshop on ML*, 2009.

[16] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. *Functional Programming Languages and Computer Architecture*, pages 124–144, 1991. . URL `http://link.springer.com/chapter/10.1007/3540543961_7`.

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

[18] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. *Mathematical Foundations of Programming Semantics*, pages 209–228, 1990. . URL `papers://cff96cb1-96b7-4b11-a3e3-f4947c1d45b9/Paper/p6382`.

[19] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[20] A. Rastogi, A. Delignat-lavaud, C. Keller, P.-y. Strub, and K. Bhargavan. Semantic Purity and Effects Reunited in F. pages 1–19, 2015.

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

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

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

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

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

[26] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types, 2011. ISSN 03621340.

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

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

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

A. Full Specification of Core Language

A.1 Syntax

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

Syntactic Sugar

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

A.2 Operational Semantics

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

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

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

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

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

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

A.3 Typing

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

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

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

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

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

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

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

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

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

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

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

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

B. Proofs about Core Language

B.1 Properties

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

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. We only treat cases T_Mu, T_CastUp and T_CastDown (since proofs of other cases are the same as λC [3]):

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

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

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

□

Lemma B.2 (Thinning). *Let* Γ *and* Γ' *be legal contexts such that* $\Gamma \subseteq \Gamma'$. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma' \vdash e : \tau$.

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

Lemma B.3 (Substitution)**.** *If* $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$ *and* $\Gamma_1 \vdash e_2 : \sigma$*, then* $\Gamma_1, \Gamma_2[x \mapsto e_2] \vdash e_1[x \mapsto e_2] : \tau[x \mapsto e_2]$.

Proof. By induction on the derivation of $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$. Let $e^* \equiv e[x \mapsto e_2]$. Then the result can be written as $\Gamma_1, \Gamma_2^* \vdash {e_1}^* : \tau^*$. We only treat cases T_MU, T_CASTUP and T_CASTDOWN. Consider the last step of derivation of the following cases:

Case T_MU: $$\frac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau \qquad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau : \star}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\mu\, y : \tau . e_1) : \tau}$$
By induction hypothesis, we have $\Gamma_1, \Gamma_2^* \vdash {e_1}^* : \tau^*$ and $\Gamma_1, \Gamma_2^* \vdash \tau^* : \star$. Then by the deviation rule, $\Gamma_1, \Gamma_2^* \vdash (\mu\, y : \tau^* . {e_1}^*) : \tau^*$. Thus we have $\Gamma_1, \Gamma_2^* \vdash (\mu\, y : \tau . e_1)^* : \tau^*$ which is just the result.

Case T_CASTUP: $$\frac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau_2 \qquad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau_1 : \star \qquad \tau_1 \longrightarrow \tau_2}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\mathsf{cast}^{\uparrow} [\tau_1]\, e_1) : \tau_1}$$
By induction hypothesis, we have $\Gamma_1, \Gamma_2^* \vdash {e_1}^* : \tau_2^*$, $\Gamma_1, \Gamma_2^* \vdash \tau_1^* : \star$ and $\tau_1 \longrightarrow \tau_2$. By the definition of substitution, we can obtain $\tau_1^* \longrightarrow \tau_2^*$ by $\tau_1 \longrightarrow \tau_2$. Then by the deviation rule, $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}^{\uparrow} [\tau_1^*]\, {e_1}^*) : \tau_1^*$. Thus we have $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}^{\uparrow} [\tau_1]\, e_1)^* : \tau_1^*$ which is just the result.

Case T_CASTDOWN: $$\frac{\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau_1 \qquad \Gamma_1, x : \sigma, \Gamma_2 \vdash \tau_2 : \star \qquad \tau_1 \longrightarrow \tau_2}{\Gamma_1, x : \sigma, \Gamma_2 \vdash (\mathsf{cast}_{\downarrow}\, e_1) : \tau_2}$$
By induction hypothesis, we have $\Gamma_1, \Gamma_2^* \vdash {e_1}^* : \tau_1^*$, $\Gamma_1, \Gamma_2^* \vdash \tau_2^* : \star$ and $\tau_1 \longrightarrow \tau_2$ thus $\tau_1^* \longrightarrow \tau_2^*$. Then by the deviation rule, $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}_{\downarrow}\, {e_1}^*) : \tau_2^*$. Thus we have $\Gamma_1, \Gamma_2^* \vdash (\mathsf{cast}_{\downarrow}\, e_1)^* : \tau_2^*$ which is just the result.

□

Lemma B.4 (Generation)**.**

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

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

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

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

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

(6) If $\Gamma \vdash (\mathsf{cast}^{\uparrow} [\tau_1]\, e) : \sigma$*, then there exist an expression* τ_2 *such that* $\Gamma \vdash e : \tau_2$, $\Gamma \vdash \tau_1 : \star$, $\tau_1 \longrightarrow \tau_2$ *and* $\sigma \equiv \tau_1$.

(7) If $\Gamma \vdash (\mathsf{cast}_{\downarrow}\, e) : \sigma$*, then there exist expressions* τ_1, τ_2 *such that* $\Gamma \vdash e : \tau_1$, $\Gamma \vdash \tau_2 : \star$, $\tau_1 \longrightarrow \tau_2$ *and* $\sigma \equiv \tau_2$.

Proof. Consider a derivation of $\Gamma \vdash e : \sigma$ for one of cases in the lemma. We can follow the process of derivation until expression e is introduced the first time. The last step of derivation can be done by

- rule T_VAR for case 1;
- rule T_APP for case 2;
- rule T_LAM for case 3;
- rule T_PI for case 4;
- rule T_MU for case 5;
- rule T_CASTUP for case 6;
- rule T_CASTDOWN for case 7.

In each case, assume the conclusion of the rule is $\Gamma' \vdash e : \tau'$ where $\Gamma' \subseteq \Gamma$ and $\tau' \equiv \sigma$. Then by inspection of used derivation rules and Lemma B.2, it can be shown that the statement of the lemma holds and is the only possible case. □

Lemma B.5 (Correctness of Types)**.** *If* $\Gamma \vdash e : \tau$ *then* $\tau \equiv \star$ *or* $\Gamma \vdash \tau : \star$.

Proof. Trivial induction on the derivation of $\Gamma \vdash e : \tau$ using Lemma B.4. □

B.2 Decidability of Type Checking

Lemma B.6 (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 no such* e'.

Proof. By induction on the structure of e:

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

Case $e = (\lambda x : \tau . e_1)\, e_2$**:** There is a unique $e' = e_1[x \mapsto e_2]$ by rule S_BETA.

Case $e = \mathsf{cast}_{\downarrow} (\mathsf{cast}^{\uparrow} [\tau]\, e)$**:** There is a unique $e' = e$ by rule S_CASTDOWNUP.

Case $e = \mu\, x : \tau . e$**:** There is a unique $e' = e[x \mapsto \mu\, x : \tau . e]$ by rule S_MU.

Case $e = e_1\, e_2$ **and** e_1 **is not a** λ**-term:** If $e_1 = v$, there is no e_1' such that $e_1 \longrightarrow e_1'$. Since e_1 is not a λ-term, there is no rule to reduce e. Thus there is no e' such that $e \longrightarrow e'$.
Otherwise, there exists some e_1' such that $e_1 \longrightarrow e_1'$. By the induction hypothesis, e_1' is unique reduction of e_1. Thus by rule S_APP, $e' = e_1'\, e_2$ is the unique reduction for e.

Case $e = \mathsf{cast}_{\downarrow}\, e_1$ **and** e_1 **is not a** $\mathsf{cast}^{\uparrow}$**-term:** If $e_1 = v$, there is no e_1' such that $e_1 \longrightarrow e_1'$. Since e_1 is not a $\mathsf{cast}^{\uparrow}$-term, there is no rule to reduce e. Thus there is no e' such that $e \longrightarrow e'$.
Otherwise, there exists some e_1' such that $e_1 \longrightarrow e_1'$. By the induction hypothesis, e_1' is unique reduction of e_1. Thus by rule S_CASTDOWN, $e' = \mathsf{cast}_{\downarrow}\, e_1'$ is the unique reduction for e.

□

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

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

Theorem B.8 (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 and τ is the unique type of x if $x : \tau \in \Gamma$.

Case $e = e_1\, e_2$**:** By rule T_APP and introduction hypothesis, there exist unique expressions τ_1 and τ_2 such that $\Gamma \vdash e_1 : (\Pi\, x : \tau_1 . \tau_2)$, $\Gamma \vdash e_2 : \tau_1$. Thus, from Lemma B.4, $\tau_2[x \mapsto e_2]$ is the unique type of e.

Case $\lambda x : \tau_1 . e_1$**:** By rule T_LAM and introduction hypothesis, there exist unique expressions τ_2 such that $\Gamma \vdash (\Pi\, x : \tau_1 . \tau_2) : \star$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$. Thus, from Lemma B.4, $\Pi\, x : \tau_1 . \tau_2$ is the unique type of e.

Case $\Pi\, x : \tau_1 . \tau_2$**:** By rule T_PI and introduction hypothesis, we have $\Gamma \vdash \tau_1 : \star$ and $\Gamma, x : \tau_1 \vdash \tau_2 : \star$. Thus, from Lemma B.4, $\star$ is the unique type of e.

Case $\mu\, x : \tau . e_1$**:** By rule T_MU and introduction hypothesis, we have $\Gamma \vdash \tau : \star$ and $\Gamma, x : \tau \vdash e : \tau$. Thus, from Lemma B.4, τ is the unique type of e.

Case $e = \mathsf{cast}^{\uparrow}\, [\tau_1]\, e_1$**:** From the premises of rule T_CASTUP, by induction hypothesis, we can derive the type of e_1 as τ_2, and check whether τ_1 is legal, i.e. its sorts is $\star$. If τ_1 is legal, by Lemma B.6, there is at most one τ_1' such that $\tau_1 \longrightarrow \tau_1'$. If such τ_1' does not exist, then we report 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 obtain the unique type of e which is τ_1. Otherwise, we report e fails to type check.

Case $e = \mathsf{cast}_{\downarrow}\, e_1$**:** From the premises of rule T_CASTDOWN, by induction hypothesis, we can derive the type of e_1 as τ_1. By Lemma B.6, there is at most one τ_2 such that $\tau_1 \longrightarrow \tau_2$. If such τ_2 exists and its sorts is $\star$, we have found the unique type of e is τ_2. Otherwise, we report e fails to type check.

□

B.3 Soundness

Definition B.9 (Multi-step reduction)**.** *The relation $\twoheadrightarrow$ is the transitive and reflexive closure of $\longrightarrow$.*

Theorem B.10 (Subject Reduction)**.** *If $\Gamma \vdash e : \sigma$ and $e \twoheadrightarrow e'$ then $\Gamma \vdash e' : \sigma$.*

Proof. We prove the case for one-step reduction, i.e. $e \longrightarrow e'$. The lemma can follow by induction on the number of one-step reductions of $e \twoheadrightarrow e'$. The proof is by induction with respect to the definition of one-step reduction $\longrightarrow$ as follows:

Case $\dfrac{}{(\lambda x : \tau . e_1)\; e_2 \longrightarrow e_1[x \mapsto e_2]}$ **S_BETA:**
Suppose $\Gamma \vdash (\lambda x : \tau_1 . e_1)\; e_2 : \sigma$ and $\Gamma \vdash e_1[x \mapsto e_2] : \sigma'$. By Lemma B.4(2), there exist expressions τ_1' and τ_2 such that

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

By Lemma B.4(3), the judgement (1) implies that there exists an expression τ_2' such that

$$\Pi\, x : \tau_1' . \tau_2 \equiv \Pi\, x : \tau_1 . \tau_2' \tag{2}$$
$$\Gamma, x : \tau_1 \vdash e_1 : \tau_2'$$

Hence, by (2) we have $\tau_1 \equiv \tau_1'$ and $\tau_2 \equiv \tau_2'$. Then we can obtain $\Gamma, x : \tau_1 \vdash e_1 : \tau_2$ and $\Gamma \vdash e_2 : \tau_1$. By Lemma B.3, we have $\Gamma \vdash e_1[x \mapsto e_2] : \tau_2[x \mapsto e_2]$. Therefore, we conclude with $\sigma' \equiv \tau_2[x \mapsto e_2] \equiv \sigma$.

Case $\dfrac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2}$ **S_APP:**
Suppose $\Gamma \vdash e_1\; e_2 : \sigma$ and $\Gamma \vdash e_1'\; e_2 : \sigma'$. By Lemma B.4(2), there exist expressions τ_1 and τ_2 such that

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

By induction hypothesis, we have $\Gamma \vdash e_1' : (\Pi\, x : \tau_1 . \tau_2)$. By rule T_APP, we obtain $\Gamma \vdash e_1'\; e_2 : \tau_2[x \mapsto e_2]$. Therefore, $\sigma' \equiv \tau_2[x \mapsto e_2] \equiv \sigma$.

Case $\dfrac{e \longrightarrow e'}{\mathsf{cast}_{\downarrow}\, e \longrightarrow \mathsf{cast}_{\downarrow}\, e'}$ **S_CASTDOWN:**
Suppose $\Gamma \vdash \mathsf{cast}_{\downarrow}\, e : \sigma$ and $\Gamma \vdash \mathsf{cast}_{\downarrow}\, e' : \sigma'$. By Lemma B.4(7), there exist expressions τ_1, τ_2 such that

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

By induction hypothesis, we have $\Gamma \vdash e' : \tau_1$. By rule T_CASTDOWN, we obtain $\Gamma \vdash \mathsf{cast}_{\downarrow}\, e' : \tau_2$. Therefore, $\sigma' \equiv \tau_2 \equiv \sigma$.

Case $\dfrac{}{\mathsf{cast}_{\downarrow}\, (\mathsf{cast}^{\uparrow}\, [\tau]\; e) \longrightarrow e}$ **S_CASTDOWNUP:**
Suppose $\Gamma \vdash \mathsf{cast}_{\downarrow}\, (\mathsf{cast}^{\uparrow}\, [\tau_1]\; e) : \sigma$ and $\Gamma \vdash e : \sigma'$. By Lemma B.4(7), there exist expressions τ_1', τ_2 such that

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

By Lemma B.4(6), the judgement (3) implies that there exists an expression τ_2' such that

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

By (4, 7, 8) and Lemma B.6 we obtain $\tau_2 \equiv \tau_2'$. From (6) we have $\sigma' \equiv \tau_2'$. Therefore, by (5), $\sigma' \equiv \tau_2' \equiv \tau_2 \equiv \sigma$.

Case $\dfrac{}{\mu\, x : \tau . e \longrightarrow e[x \mapsto \mu\, x : \tau . e]}$ **S_MU:**
Suppose $\Gamma \vdash (\mu\, x : \tau . e) : \sigma$ and $\Gamma \vdash e[x \mapsto \mu\, x : \tau . e] : \sigma'$. By Lemma B.4(5), we have $\sigma \equiv \tau$ and $\Gamma, x : \tau \vdash e : \tau$. Then we obtain $\Gamma \vdash (\mu\, x : \tau . e) : \tau$. Thus by Lemma B.3, we have $\Gamma \vdash e[x \mapsto \mu\, x : \tau . e] : \tau[x \mapsto \mu\, x : \tau . e]$.
Note that $x : \tau$, i.e. the type of x is τ, then $x \notin \mathsf{FV}(\tau)$ holds implicitly. Hence, by the definition of substitution, we obtain $\tau[x \mapsto \mu\, x : \tau . e] \equiv \tau$. Therefore, $\sigma' \equiv \tau[x \mapsto \mu\, x : \tau . e] \equiv \tau \equiv \sigma$.

□

Theorem B.11 (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 B.4(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 B.4(3), it must be a λ-term such that $e_1 \equiv \lambda x : \tau_1 . e_1'$ for some e_1' satisfying $\varnothing \vdash e_1' : \tau_2$. 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 induction hypothesis, there exists e_1' such that $e_1 \longrightarrow e_1'$. Then by rule S_APP, we have $e_1\; e_2 \longrightarrow e_1'\; e_2$. Thus, there exists $e' \equiv e_1'\; e_2$ such that $e \longrightarrow e'$.

Case $e = \mathsf{cast}_{\downarrow}\, e_1$**:** By Lemma B.4(7), there exist expressions τ_1 and τ_2 such that $\varnothing \vdash e_1 : \tau_1$ and $\tau_1 \longrightarrow \tau_2$. Consider whether e_1 is a value:

- If $e_1 = v$, by Lemma B.4(6), it must be a $\mathsf{cast}^{\uparrow}$-term such that $e_1 \equiv \mathsf{cast}^{\uparrow}\, [\tau_1]\; e_1'$ for some e_1' satisfying $\varnothing \vdash e_1' : \tau_2$. 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 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 7.

C.2 Operational Semantics

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

$$\frac{}{(\lambda x : T.E_1)\; E_2 \longrightarrow E_1[x \mapsto E_2]}\;\textsc{SC\_Beta}$$

$$\frac{}{\mathsf{cast}_\downarrow\,(\mathsf{cast}^\uparrow\,[T]\; E) \longrightarrow E}\;\textsc{SC\_CastDownUp}$$

$$\frac{E_1 \longrightarrow E_1'}{E_1\; E_2 \longrightarrow E_1'\; E_2}\;\textsc{SC\_App}$$

$$\frac{E \longrightarrow E'}{\mathsf{cast}_\downarrow\, E \longrightarrow \mathsf{cast}_\downarrow\, E'}\;\textsc{SC\_CastDown}$$

$$\frac{}{\mu\, x : T.E \longrightarrow E[x \mapsto \mu\, x : T.E]}\;\textsc{SC\_Mu}$$

$$\frac{E_1 \longrightarrow E_1'}{\textbf{case}\; E_1\; \textbf{of}\; \overline{p \Rightarrow E} \longrightarrow \textbf{case}\; E_1'\; \textbf{of}\; \overline{p \Rightarrow E}}\;\textsc{SC\_Case}$$

$$\frac{K_i\; \overline{x_i : T_i} \Rightarrow E_i \in \overline{p \Rightarrow E}}{\textbf{case}\; K_i\; \overline{u}\; \overline{E_1}\; \textbf{of}\; \overline{p \Rightarrow E} \longrightarrow_n E_i[\,\overline{x_i \mapsto E_1}\,]}\;\textsc{SC\_CaseMatch}$$

C.3 Expression Typing

See Figure 8.

C.4 Translation to the Core

See Figure 9.

D. Proofs about Surface Language

D.1 Type-safety of Translation

Lemma D.1 (Type-safety of Reduction Translation)**.** *If $E_1 \longrightarrow E_2$ and $\Sigma \vDash_s E_1 : T_1 \rightsquigarrow e_1$, $\Sigma \vDash_s E_2 : T_2 \rightsquigarrow e_2$ for some context Σ, then $e_1 \longrightarrow e_2$.*

Proof. By induction on the relation $E_1 \longrightarrow E_2$. Most cases are the same as core language, which are trivial. We only treat interesting cases SC_CASE and SC_CASEMATCH.

Case $\dfrac{E_1 \longrightarrow E_1'}{\textbf{case}\; E_1\; \textbf{of}\; \overline{p \Rightarrow E} \longrightarrow \textbf{case}\; E_1'\; \textbf{of}\; \overline{p \Rightarrow E}}$ **SC_CASE:**
By induction hypothesis, we have

$$\begin{array}{l}\Sigma \vDash_s E_1 : T \rightsquigarrow e_1 \\ \Sigma \vDash_s E_1' : T' \rightsquigarrow e_1' \\ e_1 \longrightarrow e_1'.\end{array}$$

Note that

$$\begin{array}{l}\Sigma \vDash_s \textbf{case}\; E_1\; \textbf{of}\; \overline{p \Rightarrow E}\; : T \rightsquigarrow (\mathsf{cast}^n_\downarrow\, e_1)\, \tau\; \overline{e} \\ \Sigma \vDash_s \textbf{case}\; E_1'\; \textbf{of}\; \overline{p \Rightarrow E}\; : T \rightsquigarrow (\mathsf{cast}^n_\downarrow\, e_1')\, \tau\; \overline{e}\,.\end{array}$$

By S_CASTDOWN, we have the result

$$(\mathsf{cast}^n_\downarrow\, e_1)\, \tau\; \overline{e} \longrightarrow (\mathsf{cast}^n_\downarrow\, e_1')\, \tau\; \overline{e}\,.$$

Case $\dfrac{K_i\; \overline{x_i : T_i} \Rightarrow E_i \in \overline{p \Rightarrow E}}{\textbf{case}\; K_i\; \overline{u}\; \overline{E_1}\; \textbf{of}\; \overline{p \Rightarrow E} \longrightarrow_n E_i[\,\overline{x_i \mapsto E_1}\,]}$ **SC_CASEMATCH:**
By rule TRDECL_DATA, $K_i \equiv \lambda\, \overline{u : \sigma^*}\, .\lambda\, \overline{x : \sigma}\, .\mathsf{cast}^n_\uparrow\, [D\; \overline{u}\,]\, (\lambda\alpha : \star.\lambda\, \overline{b : \overline{\sigma} \to \alpha}\, .b_i\; \overline{x}\,)$. By TRPAT_ALT and TR_CASE, we have

$$\Sigma \vDash_s \textbf{case}\; K_i\; \overline{u}\; \overline{E_1}\; \textbf{of}\; \overline{p \Rightarrow E}\; : T \rightsquigarrow (\mathsf{cast}^n_\downarrow\, (K_i\; \overline{u'}\; \overline{e_1}\,))\, \tau\; \overline{e'}$$

where

$$\begin{array}{ll}\overline{\Sigma \vDash_s u : \kappa \rightsquigarrow u'} & \overline{\Sigma \vDash_s E_1 : T_1 \rightsquigarrow e_1} \\ \overline{\Sigma \vDash_s T_1 : \star \rightsquigarrow \sigma} & \overline{\Sigma \vDash_s E : T \rightsquigarrow e} \\ \overline{e' \equiv \lambda\, \overline{x : \sigma}\, .e}\,. & \end{array}$$

Thus, we have the following reduction sequence:

$$\begin{array}{l}\mathsf{cast}^n_\downarrow\, (\mathsf{cast}^n_\uparrow\, [D\; \overline{u'}\,]\, (\lambda\alpha : \star.\lambda\, \overline{b : \overline{\sigma} \to \alpha}\, .b_i\; \overline{e_1}\,))\, \tau\; \overline{\lambda\, \overline{x : \sigma}\, .e} \\ \longrightarrow_n (\lambda\alpha : \star.\lambda\, \overline{b : \overline{\sigma} \to \alpha}\, .b_i\; \overline{e_1}\,)\, \tau\; \overline{\lambda\, \overline{x : \sigma}\, .e} \\ \longrightarrow (\lambda\, \overline{b : \overline{\sigma} \to \tau}\, .b_i\; \overline{e_1}\,)\; \overline{\lambda\, \overline{x : \sigma}\, .e} \\ \longrightarrow (\lambda\, \overline{x : \sigma}\, .e_i)\; \overline{e_1} \\ \longrightarrow e_i[\,\overline{x \mapsto e_1}\,]\,.\end{array}$$

Note that $\Sigma \vDash_s E_i[\,\overline{x_i \mapsto E_1}\,] : T \rightsquigarrow e_i[\,\overline{x_i \mapsto e_1}\,]$, therefore the reduction sequence above follows the result.

□

Theorem D.2 (Type-safety of Expression Translation)**.** *If $\Sigma \vDash_s E : T \rightsquigarrow e$, $\Sigma \vDash_s T : \star \rightsquigarrow \tau$ and $\vDash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma$, then $\Gamma \vdash e : \tau$.*

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

Case TR_AX: Trivial. $e = \tau = \star$ and $\Sigma \vDash_s \star : \star$ holds by rule T_AX.

Case TR_VAR: Trivial. By rule T_VAR, we have $\vDash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma$, then $x : \tau \in \Gamma$ where $\Sigma \vDash_s T : \star \rightsquigarrow \tau$.

Case TR_APP: Suppose

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

By induction hypothesis, we have

$$\Gamma \vdash e_1 : (\Pi\, x : \tau_2.\tau_1) \qquad \Gamma \vdash e_2 : \tau_2,$$

where

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

Thus by rule T_APP, we have $\Gamma \vdash e_1\; e_2 : \tau_1[x \mapsto e_2]$.

Case TR_LAM: Suppose

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

By induction hypothesis, we have

$$\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash \Pi\, x : \tau_1.\tau_2 : \star$$

where

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

Thus by rule T_LAM, we have $\Gamma \vdash (\lambda x : \tau_1.e) : (\Pi\, x : \tau_1.\tau_2)$.

Case TR_PI: Suppose

$$\Sigma \vDash_s (\Pi\, x : T_1.T_2) : \star \rightsquigarrow \Pi\, x : \tau_1.\tau_2.$$

pgm	$::=$		Program
	$\vert$	$\overline{decl}\,; E$	Declarations
$decl$	$::=$		Declarations
	$\vert$	$\mathbf{data}\, D\, \overline{u:\kappa} \;=\; \overline{K\, \overline{T}}$	Datatype declaration
$u,\ \upsilon$	$::=$		Atoms
	$\vert$	x	Variable
	$\vert$	K	Data constructor
p	$::=$		Patterns
	$\vert$	$K\, \overline{x:T}$	Pattern
$E,\ T,\ S,\ \kappa$	$::=$		Expressions
	$\vert$	u	Atom
	$\vert$	$\star$	Type of types
	$\vert$	$E_1\, E_2$	Application
	$\vert$	$\lambda x : T.E$	Abstraction
	$\vert$	$\Pi\, x : T.S$	Product
	$\vert$	$\mathsf{cast}^{\uparrow}\, [T]\, E$	Cast up by expansion
	$\vert$	$\mathsf{cast}_{\downarrow}\, E$	Cast down by reduction
	$\vert$	$\mu\, x : T.E$	General recursion
	$\vert$	$\mathbf{case}\, E_1\, \mathbf{of}\, \overline{p \Rightarrow E_2}$	Case analysis
v	$::=$		Values
	$\vert$	$\star$	Type of types
	$\vert$	$\lambda x : T.E$	Abstraction
	$\vert$	$\Pi\, x : T.S$	Product
	$\vert$	$\mathsf{cast}^{\uparrow}\, [T]\, E$	Cast up by expansion
	$\vert$	$K\, \overline{u}\, \overline{E}$	Data constructor
Σ	$::=$		Contexts
	$\vert$	$\varnothing$	Empty
	$\vert$	$\Sigma, u : T$	Atom binding

				Syntactic Sugar
$T_1 \to T_2$	$\triangleq$	$\Pi\, x : T_1.T_2$	$x \notin \mathsf{FV}(T_2)$	Function type
$(x : T_1) \to T_2$	$\triangleq$	$\Pi\, x : T_1.T_2$	$x \in \mathsf{FV}(T_2)$	Dependent function type
$\mathsf{cast}^{n}_{\uparrow}\, [T_1]\, E$	$\triangleq$	$\mathsf{cast}^{\uparrow}[T_1](\mathsf{cast}^{\uparrow}[T_2](\ldots(\mathsf{cast}^{\uparrow}\, [T_n]\, E)\ldots))$		Cast up in n steps
$\mathsf{cast}^{n}_{\downarrow}\, E$	$\triangleq$	$\underbrace{\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow}(\ldots(\mathsf{cast}_{\downarrow}}_{n}\, E)\ldots))$		Cast down in n steps
$\mathbf{data}\, R\, \overline{u:\kappa} \;=\; K\{\, \overline{N:T}\, \}$	$\triangleq$	$\mathbf{data}\, R\, \overline{u:\kappa} \;=\; K\, \overline{T}\,;$ $\mathbf{let}\ N_i : (\, \overline{u:\kappa}\,) \to R\, \overline{u} \;\to T_i =$ $\lambda\, \overline{u:\kappa}\,.\lambda y : R\, \overline{u}\,.\mathbf{case}\, y\, \mathbf{of}\, K\, \overline{x:T} \;\Rightarrow x_i\ \mathbf{in}$	$\forall i$	Record

Figure 7. Syntax of the surface language

By induction hypothesis, we have

$$\Gamma \vdash \tau_1 : \star \qquad \Gamma, x : \tau_1 \vdash \tau_2 : \star$$

where

$$\Sigma \vDash_{\mathsf{s}} T_1 : \star \rightsquigarrow \tau_1 \qquad \Sigma, x : T_1 \vDash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2$$

Thus by rule T_PI $\Gamma \vdash (\Pi\, x : \tau_1.\tau_2) : \star$.

Case TR_CASTUP: Suppose

$$\Sigma \vDash_{\mathsf{s}} (\mathsf{cast}^{\uparrow}\, [T_1]\, E) : T_1 \rightsquigarrow \mathsf{cast}^{\uparrow}\, [\tau_1]\, e$$
$$\Sigma \vDash_{\mathsf{s}} T_1 : \star \rightsquigarrow \tau_1.$$

By induction hypothesis, we have

$$\Gamma \vdash e : \tau_2 \qquad \Sigma \vDash_{\mathsf{s}} T_1 : \star \rightsquigarrow \tau_1$$
$$T_1 \longrightarrow T_2,$$

where

$$\Sigma \vDash_{\mathsf{s}} E : T_2 \rightsquigarrow e \quad \Sigma \vDash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2.$$

By Lemma D.1, we can obtain $\tau_1 \longrightarrow \tau_2$. Thus, by rule T_CASTUP, we have $\Gamma \vdash \mathsf{cast}^{\uparrow}\, [\tau_1]\, e : \tau_1$.

Case TR_CASTDOWN: Suppose

$$\Sigma \vDash_{\mathsf{s}} (\mathsf{cast}_{\downarrow}\, E) : T_2 \rightsquigarrow \mathsf{cast}_{\downarrow}\, e \qquad \Sigma \vDash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2.$$

By induction hypothesis, we have

$$\Gamma \vdash e : \tau_1 \qquad T_1 \longrightarrow T_2$$

where

$$\Sigma \vDash_{\mathsf{s}} E : T_1 \rightsquigarrow e$$
$$\Sigma \vDash_{\mathsf{s}} T_1 : \star \rightsquigarrow \tau_1 \quad \Sigma \vDash_{\mathsf{s}} T_2 : \star \rightsquigarrow \tau_2.$$

By Lemma D.1, we obtain $\tau_1 \longrightarrow \tau_2$. Thus, by rule T_CASTDOWN, we have $\Gamma \vdash \mathsf{cast}_{\downarrow}\, e : \tau_2$.

Case TR_MU: Suppose

$$\Sigma \vDash_{\mathsf{s}} (\mu\, x : T.E) : T \rightsquigarrow \mu\, x : \tau.e$$
$$\Sigma \vDash_{\mathsf{s}} T : \star \rightsquigarrow \tau.$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Sigma \vdash_{\mathsf{s}} E : T_1 \qquad \Sigma \vdash_{\mathsf{s}} T_2 : \star \qquad T_1 \longrightarrow T_2}{\Sigma \vdash_{\mathsf{s}} (\mathsf{cast}_{\downarrow}\, E) : T_2} \quad \textsc{TS\_CastDown}$$

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

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

Figure 8. Typing rules of the surface language

$\boxed{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma}$ Context translation

$$\frac{}{\vdash_{\mathsf{wf}} \varnothing \rightsquigarrow \varnothing} \quad \text{TRENV\_EMPTY}$$

$$\frac{\vdash_{\mathsf{wf}} \Sigma \rightsquigarrow \Gamma \qquad \Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau}{\vdash_{\mathsf{wf}} \Sigma, x : T \rightsquigarrow \Gamma, x : \tau} \quad \text{TRENV\_VAR}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 9. Translation rules of the surface language

By 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 have $\Gamma \vdash (\mu\, x : \tau . e) : \tau$.

Case TR_CASE: Suppose

$$\begin{array}{l} \Sigma \vdash_{\mathsf{s}} \textbf{case } E_1 \textbf{ of } \overline{p \Rightarrow E_2} : T \rightsquigarrow (\mathsf{cast}^n_{\downarrow}\, e_1)\, \tau\, \overline{e_2} \\ \Sigma \vdash_{\mathsf{s}} T : \star \rightsquigarrow \tau. \end{array}$$

By induction hypothesis, we have

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

By rule TRPAT_ALT, we have

$$\begin{aligned} p &\equiv K\, \overline{x : S_1[\,\overline{u \mapsto \upsilon}\,]} \\ S &\equiv D\, \overline{\upsilon} \\ e_2 &\equiv \lambda\, \overline{x : \sigma}\, .e \end{aligned}$$

where

$$\begin{array}{l} \Sigma \vdash_{\mathsf{s}} E_2 : T \rightsquigarrow e \\ \Gamma \vdash e : \tau \\ \Sigma \vdash_{\mathsf{s}} S_1[\,\overline{u \mapsto \upsilon}\,] : \star \rightsquigarrow \sigma \end{array}$$

By rule TRDECL_DATA, we have $D \equiv \mu X : (\,\overline{u : \sigma^*}\,) \to \star . \lambda\, \overline{u : \sigma^*}\, .(\alpha : \star) \to \overline{(\,\overline{\sigma}\,[D \mapsto X] \to \alpha)} \to \alpha$. Thus,

$$\tau_1 \equiv D\, \overline{\sigma^*}\,, \text{ where } \overline{\Sigma \vdash_{\mathsf{s}} \upsilon : \star \rightsquigarrow \sigma^*}\,.$$

Then by rule T_CASTDOWN and the definition of n-step cast operator, the type of $\mathsf{cast}^n_{\downarrow}\, e_1$ is

$$(\alpha : \star) \to \overline{(\,\overline{\sigma} \to \alpha)} \to \alpha.$$

Note that by rule T_LAM, $\Gamma \vdash e_2 : \overline{\sigma} \to \tau$. Therefore, by rule T_APP, we obtain $\Gamma \vdash (\mathsf{cast}^n_{\downarrow}\, e_1)\, \tau\, \overline{e_2} : \tau$, which follows the result.

□