Unified Syntax with Iso-Types

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

The University of Hong Kong, Pokfulam, Hong Kong, China
{ypyang,xbi,bruno}@cs.hku.hk

Abstract. Traditional designs for functional languages (such as Haskell or ML) have separate sorts of syntax for terms and types. In contrast, many dependently typed languages use a unified syntax that accounts for both terms and types. Unified syntax has some interesting advantages over separate syntax, including less duplication of concepts, and added expressiveness. However, integrating *unrestricted* general recursion in calculi with unified syntax is challenging when some level of type-level computation is present, as *decidable type-checking* is easily lost.

This paper argues that the advantages of unified syntax also apply to traditional functional languages, and there is no need to give up decidable type-checking. We present a dependently typed calculus that uses unified syntax, supports general recursion and has decidable type-checking. The key to retain decidable type-checking is a generalization of *iso-recursive types* called *iso-types*. Iso-types replace the conversion rule typically used in dependently typed calculus, and make every computation explicit via cast operators. We study two variants of the calculus that differ on the reduction strategy employed by the cast operators, and give different trade-offs in terms of simplicity and expressiveness.

1 Introduction

We live exciting times in the design of functional programming languages. In recent years, dependent types have inspired novel designs for new programming languages, such as Agda [19] or Idris [6], as well as numerous programming language research [2, 3, 8, 23–25, 28]. Dependently typed languages bring additional expressiveness to type systems, and they can also support different forms of assurances, such as strong normalization and logical consistency, not typically present in traditional programming languages. Nevertheless, traditional designs for functional languages still have some benefits. While strong normalization and logical consistency are certainly nice properties to have, and can be valuable to have in many domains, they can also impose restrictions on how programs are written. For example, the termination checking algorithms typically employed by dependently typed languages such as Agda or Idris can only automatically ensure termination of programs that follow certain patterns. In contrast Haskell or ML programmers can write their programs much more freely, since they do not need to worry about retaining strong normalization and logical consistency. Thus there is still plenty of space for both types of designs.

From an implementation and foundational point-of-view, dependently typed languages and traditional functional languages also have important differences. Languages like Haskell or ML have a strong separation between terms and types (and also kinds). This separation often leads to duplication of constructs. For example, when the type language provides some sort of type level computation, constructs such as type application (mimicking value level application) may be needed. In contrast many dependently typed languages unify types and terms. There are benefits in unifying types and terms. In addition to the extra expressiveness afforded, for example, by dependent types, only one syntactic level is needed. Thus duplication can be avoided. Having less language constructs simplifies the language, making it easier to study (from the meta-theoretical point of view) and maintain (from the implementation point of view).

In principle having unified syntax would be beneficial even for more traditional designs of functional languages, which have no strong normalization or logical consistency. Not surprisingly, researchers have in the past considered such an option for implementing functional languages [3, 7, 21], by using some variant of *pure type systems* (PTS) [5] (normally extended with general recursion). Thus, with a simple and tiny calculus, they showed that powerful and quite expressive functional languages could be built with unified syntax.

However having unified syntax for types and terms brings challenges. One pressing problem is that integrating (unrestricted) general recursion in dependently typed calculi with unified syntax, while retaining logical consistency, strong normalization and *decidable type-checking* is difficult. Indeed, many early designs using unified syntax and unrestricted general recursion [3, 7] lose all three properties. For pragmatic reasons languages like Agda or Idris also allow turning off the termination checker, which allows for added expressiveness, but loses the three properties as well. More recently, various researchers [8, 24, 27] have been investigating how to combine those properties, general recursion and dependent types. However, this is usually done by having the type system carefully control the total and partial parts of computation, adding significant complexity to those calculi when compared to systems based on pure type systems.

Nevertheless, if we are interested in traditional languages, only the loss of decidable type-checking is problematic. Unlike strong normalization and logical consistency, decidable type-checking is normally one property that is expected from a traditional programming language design.

This paper proposes λI: a simple call-by-name variant of the calculus of constructions. The key challenge solved in this work is how to define a calculus comparable in simplicity to the calculus of constructions, while featuring both general recursion and decidable type checking. The main idea, is to recover decidable type-checking by making each type-level computation step explicit. In essence, each type-level reduction or expansion is controlled by a *type-safe* cast. Since single computation steps are trivially terminating, decidability of type checking is possible even in the presence of non-terminating programs at the type level. At the same time term-level programs using general recursion work as in any conventional functional languages, and can be non-terminating.

Our design generalizes *iso-recursive types* [22], which are our main source of inspiration. In λI, not only folding/unfolding of recursion at the type level is explicitly controlled by term level constructs, but also any other type level computation (including beta reduction/expansion). There is an analogy to language designs with *equi-recursive* types and *iso-recursive* types, which are normally the two options for adding recursive types to languages. With equi-recursive types, type-level recursion is implicitly folded/unfolded, which makes establishing decidability of type-checking much more difficult. In iso-recursive designs, the idea is to trade some convenience by a simple way to ensure decidability. Similarly, we view the design of traditional dependently typed calculi, such as the calculus of constructions as analogous to systems with equi-recursive types. In the calculus of constructions it is the *conversion rule* that allows type-level computation to by implicitly triggered. However, the proof of decidability of type checking for the *calculus of constructions* [10] (and other normalizing PTS) is non-trivial, as it depends on strong normalization [16]. Moreover decidability is lost when adding general recursion. In contrast, the cast operators in λI have to be used to *explicitly* trigger each step of type-level computation, but it is easy to ensure decidable type-checking, even with general recursion.

We study two variants of the calculus that differ on the reduction strategy employed by the cast operators, and give different trade-offs in terms of simplicity and expressiveness. The first variant λI_{w} uses weak-head reduction in the cast operators. This allows for a very simple calculus, but loses some expressiveness in terms of type level computation. Nevertheless in this variant it is still possible to encode useful language constructs such as algebraic datatypes. The second variant λI_{p} uses *parallel reduction* for casts and is more expressive. It allows equating terms such as $\mathit{Vec}\ (1 + 1)$ and $\mathit{Vec}\ 2$ as equal. The price to pay for this more expressive design is some additional complexity. For both designs type soundness and decidability of type-checking are proved.

It is worth emphasizing that λI does sacrifice some convenience when performing type-level computations in order to gain the ability of doing arbitrary general recursion at the term level. The goal of this work is to show the benefits of unified syntax in terms of economy of concepts for programming language design, and not use unified syntax to express computationally intensive type-level programs. Investigating how to express computationally intensive type-level programs (as in dependently typed programming) in λI is left for future work.

In summary, the contributions of this work are:

- **The λI calculus:** A simple calculus for functional programming, that collapses terms, types and kinds into the same hierarchy and supports general recursion. λI is type-safe and the type system is decidable. Full proofs are provided in the extended version of this paper [29].
- **Iso-types:** λI generalizes iso-recursive types by making all type-level computation steps explicit via *casts operators*. In λI the combination of casts and recursion subsumes iso-recursive types.
- **A prototype implementation:** The prototype of λI is available online[1].

[1] `https://bitbucket.org/ypyang/aplas16`

2 Overview

In this section, we informally introduce the main features of λI. In particular, we show how the casts in λI can be used instead of the typical conversion rule present in calculi such as the calculus of constructions. The formal details of λI are presented in Section 3 and 4.

2.1 The Calculus of Constructions and the Conversion Rule

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

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

It allows one to derive $e : \tau_2$ from the derivation of $e : \tau_1$ and the beta equality of τ_1 and τ_2. This rule is important to *automatically* allow terms with beta equivalent types to be considered type-compatible. For example, consider the following identity function:

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

The type of y is a *type-level* identity function applied to *Int*. Without the conversion rule, f cannot be applied to 3 for example, since the type of 3 (*Int*) differs from the type of y ($(\lambda x : \star.\ x)\ \mathit{Int}$). Note that the beta equivalence $(\lambda x : \star.\ x)\ \mathit{Int} =_\beta \mathit{Int}$ holds. The conversion rule allows the application of f to 3 by converting the type of y to *Int*.

Decidability of Type Checking and Strong Normalization. While the conversion rule in λC brings a lot of convenience, an unfortunate consequence is that it couples decidability of type checking with strong normalization of the calculus [16]. Therefore adding general recursion to λC becomes difficult, since strong normalization is lost. Due to the conversion rule, any non-terminating term would force the type checker to go into an infinite loop (by constantly applying the conversion rule without termination), thus rendering the type system undecidable. For example, assume a term z that has type loop, where loop stands for any diverging computation. If we type check $(\lambda x : \mathit{Int}.\ x)\ z$ under the normal typing rules of λC, the type checker would get stuck as it tries to do beta equality on two terms: *Int* and loop, where the latter is non-terminating.

2.2 An Alternative to the Conversion Rule: Iso-Types

In contrast to the conversion rule of λC, λI features *iso-types*, making it explicit as to when and where to convert one type to another. Type conversions are explicitly controlled by two language constructs: $\mathsf{cast}_\downarrow$ (one-step reduction) and $\mathsf{cast}_\uparrow$ (one-step expansion). The benefit of this approach is that decidability of type checking is no longer coupled with strong normalization of the calculus.

Reduction. The $\mathsf{cast}_{\downarrow}$ operator allows a type conversion provided that the resulting type is a *reduction* of the original type of the term. To explain the use of $\mathsf{cast}_{\downarrow}$, assume an identity function g defined by $g = \lambda y : \mathit{Int}.\ y$ and a term e such that $e : (\lambda x : \star.\ x)\,\mathit{Int}$. In contrast to λC, we cannot directly apply g to e in λI since the type of e $((\lambda x : \star.\ x)\,\mathit{Int})$ is not *syntactically equal* to Int. However, note that the reduction relation $(\lambda x : \star.\ x)\,\mathit{Int} \longrightarrow \mathit{Int}$ holds. We can use $\mathsf{cast}_{\downarrow}$ for the explicit (type-level) reduction:

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

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

Expansion. The dual operation of $\mathsf{cast}_{\downarrow}$ is $\mathsf{cast}_{\uparrow}$, which allows a type conversion provided that the resulting type is an *expansion* of the original type of the term. To explain the use of $\mathsf{cast}_{\uparrow}$, let us revisit the example from Section 2.1. We cannot apply f to 3 without the conversion rule. Instead, we can use $\mathsf{cast}_{\uparrow}$ to expand the type of 3:

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

Thus, the application $f\ (\mathsf{cast}_{\uparrow}\,[(\lambda x : \star.\ x)\,\mathit{Int}]\,3)$ becomes well-typed. Intuitively, $\mathsf{cast}_{\uparrow}$ performs expansion, as the type of 3 is Int, and $(\lambda x : \star.\ x)\,\mathit{Int}$ is the expansion of Int witnessed by $(\lambda x : \star.\ x)\,\mathit{Int} \longrightarrow \mathit{Int}$. Notice that for $\mathsf{cast}_{\uparrow}$ to work, we need to provide the resulting type as argument. This is because for the same term, there may be more than one choice for expansion. For example, $1+2$ and $2+1$ are both the expansions of 3.

One-Step. The cast rules allow only *one-step* reduction or expansion. If two type-level terms require more than one step of reductions or expansions for normalization, then multiple casts must be used. Consider a variant of the example such that $e : (\lambda x : \star.\ \lambda y : \star.\ x)\,\mathit{Int}\,\mathit{Bool}$. Given $g = \lambda y : \mathit{Int}.\ y$, the expression $g\ (\mathsf{cast}_{\downarrow}\ e)$ is ill-typed because $\mathsf{cast}_{\downarrow}\ e$ has type $(\lambda y : \star.\ \mathit{Int})\,\mathit{Bool}$, which is not syntactically equal to Int. Thus, we need another $\mathsf{cast}_{\downarrow}$:

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

to further reduce the type and allow the program $g\ (\mathsf{cast}_{\downarrow}\,(\mathsf{cast}_{\downarrow}\ e))$ to type check.

Decidability without Strong Normalization. With explicit type conversion rules the decidability of type checking no longer depends on the strong normalization property. Thus the type system remains decidable even in the presence of non-termination at type level. Consider the same example using the term z from Section 2.1. This time the type checker will not get stuck when type checking $(\lambda x : \mathit{Int}.\ x)\,z$. This is because in λI, the type checker only performs syntactic comparison between Int and loop, instead of beta equality. Thus it rejects the above application as ill-typed. Indeed it is impossible to type check such application even with the use of $\mathsf{cast}_{\uparrow}$ and/or $\mathsf{cast}_{\downarrow}$: one would need to write infinite number of $\mathsf{cast}_{\downarrow}$'s to make the type checker loop forever (e.g., $(\lambda x : \mathit{Int}.\ x)(\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow} \ldots z))$). But it is impossible to write such program in practice.

Variants of Casts. A reduction relation is used in cast operators to convert types. We study two possible reduction relations: call-by-name *weak-head reduction* and *full reduction*. Weak-head reduction cannot reduce sub-terms at certain positions (e.g., inside λ or Π binders), while full reduction can reduce sub-terms at any position. We define two variants of casts, namely *weak* and *full* casts, by employing weak-head and full reduction respectively. We also create two variants of λI, namely λI_{w} and λI_{p}. The only difference is that λI_{w} uses weak-head reduction in weak cast operators $\mathsf{cast}_{\uparrow}$ and $\mathsf{cast}_{\downarrow}$, while λI_{p} uses full reduction, specifically *parallel reduction*, in full cast operators $\mathsf{cast}_{\Uparrow}$ and $\mathsf{cast}_{\Downarrow}$. Both variants reflect the idea of iso-types, but have trade-offs between simplicity and expressiveness: λI_{w} uses the same call-by-name reduction for both casts and evaluation to keep the system and metatheory simple, but loses some expressiveness, e.g. cannot convert $\mathit{Vec}\ (1+1)$ to $\mathit{Vec}\ 2$. λI_{p} is more expressive but results in a more complicated metatheory (see Section 4.2). Note that when generally referring to λI, we do not specify the reduction strategy, which could be either variant.

2.3 General Recursion

λI supports general recursion and allows writing unrestricted recursive programs at term level. The recursive construct is also used to model recursive types at type level. Recursive terms and types are represented by the same μ primitive.

Recursive Terms. The primitive $\mu x : \tau.\ e$ can be used to define recursive functions. For example, the factorial function would be written as:

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

We treat the μ operator as a *fixpoint*, which evaluates $\mu x : \tau.\ e$ to its recursive unfolding $e[x \mapsto \mu x : \tau.\ e]$. Term-level recursion in λI works as in any standard functional language, e.g., *fact* 3 produces 6 as expected (see Section 3.4).

Recursive Types. The same μ primitive is used at the type level to represent iso-recursive types [11]. In the *iso-recursive* approach a recursive type and its unfolding are different, but isomorphic. The isomorphism is witnessed by two operations, typically called fold and unfold. In λI, such isomorphism is witnessed by $\mathsf{cast}_{\uparrow}$ and $\mathsf{cast}_{\downarrow}$. In fact, $\mathsf{cast}_{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ *generalize* fold and unfold: they can convert any types, not just recursive types, as we shall see in the example of encoding parametrized datatypes in Section 5.

3 Dependent Types with Iso-Types

In this section, we present the λI_{w} calculus, which uses a (call-by-name) weak-head reduction strategy in casts. This calculus is very close to the calculus of constructions, except for three key differences: 1) the absence of the $\Box$ constant (due to use of the "type-in-type" axiom); 2) the existence of two cast operators;

3) general recursion on both term and type level. Unlike λC the proof of decidability of type checking for λI_{w} does not require the strong normalization of the calculus. Thus, the addition of general recursion does not break decidable type checking. In the rest of this section, we demonstrate the syntax, operational semantics, typing rules and metatheory of λI_{w}. Full proofs of the meta-theory can be found in the extended version of this paper [29].

3.1 Syntax

Figure 1 shows the syntax of λI_{w}, including expressions, contexts and values. λI_{w} uses a unified representation for different syntactic levels by following the *pure type system* (PTS) representation of λC [5]. There is no syntactic distinction between terms, types or kinds. We further merge types and kinds together by including only a single sort $\star$ instead of two distinct sorts $\star$ and $\Box$ of λC. This design brings economy for type checking, since one set of rules can cover all syntactic levels. We use metavariables τ and σ for an expression on the type-level position and e for one on the term level. We use $\tau_1 \to \tau_2$ as a syntactic sugar for $\Pi x : \tau_1.\ \tau_2$ if x does not occur free in τ_2.

Explicit Type Conversion. We introduce two new primitives $\mathsf{cast}_{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ (pronounced as "cast up" and "cast down") to replace the implicit conversion rule of λC with *one-step* explicit type conversions. The type-conversions perform two directions of conversion: $\mathsf{cast}_{\downarrow}$ is for the *one-step reduction* of types, and $\mathsf{cast}_{\uparrow}$ is for the *one-step expansion*. The $\mathsf{cast}_{\uparrow}$ construct takes a type parameter τ as the result type of one-step expansion for disambiguation (see also Section 2.2). The $\mathsf{cast}_{\downarrow}$ construct does not need a type parameter, because the result type of one-step reduction is uniquely determined, as we shall see in Section 3.5.

We use syntactic sugar $\mathsf{cast}^n_{\uparrow}$ and $\mathsf{cast}^n_{\downarrow}$ to denote n consecutive cast operators (see Figure 1). Alternatively, we can introduce them as built-in operators but treat one-step casts as syntactic sugar instead. Making n-step casts built-in can reduce the number of individual cast constructs, but makes cast operators less fundamental in the discussion of meta-theory. Thus, in the paper, we treat n-step casts as syntactic sugar but make them built-in in the implementation for better performance. Note that $\mathsf{cast}^n_{\uparrow}$ is simplified to take just one type parameter, i.e., the last type τ_1 of the n cast operations. Due to the determinacy of one-step reduction (see Lemma 1), the intermediate types can be uniquely determined, thus can be left out from the $\mathsf{cast}^n_{\uparrow}$ operator.

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

$$
\begin{array}{lrcl}
\text{Expressions} & e, \tau, \sigma & ::= & x \mid \star \mid e_1\ e_2 \mid \lambda x : \tau.\ e \mid \Pi x : \tau_1.\ \tau_2 \\
& & \mid & \mu x : \tau.\ e \mid \mathsf{cast}_{\uparrow}\,[\tau]\ e \mid \mathsf{cast}_{\downarrow}\ e \\
\text{Contexts} & \Gamma & ::= & \varnothing \mid \Gamma, x : \tau \\
\text{Values} & v & ::= & \star \mid \lambda x : \tau.\ e \mid \Pi x : \tau_1.\ \tau_2 \mid \mathsf{cast}_{\uparrow}\,[\tau]\ v \\
\text{Syntactic Sugar} & & & \\
\quad \tau_1 \to \tau_2 & \triangleq & \Pi x : \tau_1.\ \tau_2, \text{ where } x \notin \mathsf{FV}(\tau_2) & \\
\quad \mathsf{cast}^n_{\uparrow}\,[\tau_1]\ e & \triangleq & \mathsf{cast}_{\uparrow}[\tau_1](\mathsf{cast}_{\uparrow}[\tau_2](\ldots(\mathsf{cast}_{\uparrow}\,[\tau_n]\ e)\ldots)) & \\
& & \text{, where } \tau_1 \longrightarrow \tau_2 \longrightarrow \cdots \longrightarrow \tau_n & \\
\quad \mathsf{cast}^n_{\downarrow}\ e & \triangleq & \underbrace{\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow}(\ldots(\mathsf{cast}_{\downarrow}}_{n}\ e)\ldots)) &
\end{array}
$$

Fig. 1. Syntax of λI_{w}

$\boxed{e \longrightarrow e'}$ One-step Weak-head Reduction

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

$$
\frac{}{\mu x : \tau.\ e \longrightarrow e[x \mapsto \mu x : \tau.\ e]}\ \textsc{S\_Mu} \qquad \frac{e \longrightarrow e'}{\mathsf{cast}_{\downarrow}\ e \longrightarrow \mathsf{cast}_{\downarrow}\ e'}\ \textsc{S\_CastDown}
$$

$$
\frac{e \longrightarrow e'}{\mathsf{cast}_{\uparrow}\,[\tau]\ e \longrightarrow \mathsf{cast}_{\uparrow}\,[\tau]\ e'}\ \textsc{S\_CastUp} \qquad \frac{}{\mathsf{cast}_{\downarrow}\,(\mathsf{cast}_{\uparrow}\,[\tau]\ v) \longrightarrow v}\ \textsc{S\_CastElim}
$$

Fig. 2. Operational semantics of λI_{w}

3.2 Operational Semantics

Figure 2 shows the small-step, *call-by-name* operational semantics. Three base cases include S_Beta for beta reduction, S_Mu for recursion unrolling and S_CastElim for cast canceling. Three inductive cases, S_App, S_CastDown and S_CastUp, define reduction at the head position of an application, and the inner expression of $\mathsf{cast}_{\downarrow}$ and $\mathsf{cast}_{\uparrow}$ terms, respectively. Note that S_CastElim and S_CastDown do not overlap because in the former rule, the inner term of $\mathsf{cast}_{\downarrow}$ is a value (see Figure 1), i.e., $\mathsf{cast}_{\uparrow}\,[\tau]\ v$, but not a value in the latter rule.

The reduction rules are called *weak-head* because only the head term of an application can be reduced, as indicated by the rule S_App. Reduction is also not allowed inside the λ-term and Π-term which are both defined as values. Weak-head reduction rules are used for both type conversion and term evaluation. Thus, we refer to cast operators in λI_{w} as *weak* casts. To evaluate the value of a term-level expression, we apply the one-step (weak-head) reduction multiple times, i.e., multi-step reduction, the transitive and reflexive closure of the one-step reduction.

3.3 Typing

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

Most typing rules are quite standard. We write $\vdash \Gamma$ if a context Γ is well-formed. We use $\Gamma \vdash \tau : \star$ to check if τ is a well-formed type. Rule T_Ax is the "type-in-type" axiom. Rule T_Var checks the type of variable x from the valid context. Rules T_App and T_Lam check the validity of application and abstraction respectively. Rule T_Pi checks the type well-formedness of the dependent function. Rule T_Mu checks the validity of a recursive term. It ensures that the recursion $\mu x : \tau.\ e$ should have the same type τ as the binder x and also the inner expression e.

The Cast Rules. We focus on the rules T_CastUp and T_CastDown that define the semantics of cast operators and replace the conversion rule of λC. The relation between the original and converted type is defined by one-step weak-head reduction (see Figure 2). For example, given a judgment $\Gamma \vdash e : \tau_2$ and relation $\tau_1 \longrightarrow \tau_2 \longrightarrow \tau_3$, $\mathsf{cast}_{\uparrow}\,[\tau_1]\,e$ expands the type of e from τ_2 to τ_1, while $\mathsf{cast}_{\downarrow}\,e$ reduces the type of e from τ_2 to τ_3. We can formally give the typing derivations of the examples in Section 2.2:

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

Importantly, in λI_{w} 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 the normal form, then it will require n casts to compute a type-level value.

Pros and Cons of Type in Type. The "type-in-type" axiom is well-known to give rise to logical inconsistency [14]. However, since our goal is to investigate core languages for languages that are logically inconsistent anyway (due to general recursion), we do not view "type-in-type" as a problematic rule. On the other hand the rule T_Ax brings additional expressiveness and benefits: for example *kind polymorphism* [30] is supported in λI_{w}.

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

3.4 The Two Faces of Recursion

Term-level Recursion. In λI_{w}, the μ-operator works as a *fixpoint* on the term level. By rule S_Mu, evaluating a term $\mu x : \tau.\ e$ will substitute all x's in e with the whole μ-term itself, resulting in the unrolling $e[x \mapsto \mu x : \tau.\ e]$. The μ-term is equivalent to a recursive function that should be allowed to unroll without

$\boxed{\vdash \Gamma}$ Well-formed Context $\quad \dfrac{}{\vdash \varnothing}\ \textsc{Env\_Empty} \quad \dfrac{\vdash \Gamma \qquad \Gamma \vdash \tau : \star}{\vdash \Gamma, x : \tau}\ \textsc{Env\_Var}$

$\boxed{\Gamma \vdash e : \tau}$ Typing $\quad \dfrac{\vdash \Gamma}{\Gamma \vdash \star : \star}\ \textsc{T\_Ax} \quad \dfrac{\vdash \Gamma \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{T\_Var}$

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

$$\frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma, x : \tau_1 \vdash \tau_2 : \star}{\Gamma \vdash \Pi x : \tau_1.\ \tau_2 : \star}\ \textsc{T\_Pi} \quad \frac{\Gamma, x : \tau \vdash e : \tau \qquad \Gamma \vdash \tau : \star}{\Gamma \vdash \mu x : \tau.\ e : \tau}\ \textsc{T\_Mu}$$

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

Fig. 3. Typing rules of λI_{w}

restriction. Recall the factorial function example in Section 2.3. By rule T_Mu, the type of *fact* is $Int \to Int$. Thus we can apply *fact* to an integer. Note that by rule S_Mu, *fact* will be unrolled to a λ-term. Assuming the evaluation of **if-then-else** construct and arithmetic expressions follows the one-step reduction, we can evaluate the term *fact* 3 as follows:

$$\begin{array}{ll} & \mathit{fact}\ 3 \\ \longrightarrow & (\lambda x : \mathit{Int}.\ \textbf{if}\ x == 0\ \textbf{then}\ 1\ \textbf{else}\ x \times \mathit{fact}\ (x-1))\ 3 \quad \texttt{-- by S\_App} \\ \longrightarrow & \textbf{if}\ 3 == 0\ \textbf{then}\ 1\ \textbf{else}\ 3 \times \mathit{fact}\ (3-1) \quad \texttt{-- by S\_Beta} \\ \longrightarrow & \cdots \longrightarrow 6 \end{array}$$

Note that we never check if a μ-term can terminate or not, which is an undecidable problem for general recursive terms. The factorial function example above can stop, while there exist some terms that will loop forever. However, term-level non-termination is only a runtime concern and does not block the type checker. In Section 3.5 we show type checking λI_{w} is still decidable in the presence of general recursion.

Type-level Recursion. On the type level, $\mu x : \tau.\ e$ works as a *iso-recursive* type [11], a kind of recursive type that is not equal but only *isomorphic* to its unrolling. Normally, we need to add two more primitives fold and unfold for the iso-recursive type to map back and forth between the original and unrolled form. Assuming there exist expressions e_1 and e_2 such that $e_1 : \mu x : \tau.\ \sigma$ and $e_2 : \sigma[x \mapsto \mu x : \tau.\ \sigma]$, we have the following typing results:

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

by applying standard typing rules of iso-recursive types [22]:

$$\frac{\Gamma \vdash e_1 : \mu x : \tau.\ \sigma}{\Gamma \vdash \mathsf{unfold}\ e_1 : \sigma[x \mapsto \mu x : \tau.\ \sigma]} \qquad \frac{\Gamma \vdash \mu x : \tau.\ \sigma : \star \qquad \Gamma \vdash e_2 : \sigma[x \mapsto \mu x : \tau.\ \sigma]}{\Gamma \vdash \mathsf{fold}\ [\mu x : \tau.\ \sigma]\ e_2 : \mu x : \tau.\ \sigma}$$

However, in λI_{w} we do not need to introduce fold and unfold operators, because with the rule S_Mu, $\mathsf{cast}_{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ *generalize* fold and unfold. Consider the same expressions e_1 and e_2 above. The type of e_2 is the unrolling of e_1's type, which follows the one-step reduction relation by rule S_Mu: $\mu x : \tau.\ \sigma \longrightarrow \sigma[x \mapsto \mu x : \tau.\ \sigma]$. By applying rules T_CastUp and T_CastDown, we can obtain the following typing results:

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

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

An important remark is that casts are necessary, not only for controlling the unrolling of recursive types, but also for type conversion of other constructs, which is essential for encoding parametrized algebraic datatypes (see Section 5). Also, the "type-in-type" axiom [7] makes it possible to encode fixpoints even without a fixpoint primitive, i.e., the μ-operator. Thus if no casts would be performed on terms without recursive types, it would still be possible to build a term with a non-terminating type and make type-checking non-terminating.

3.5 Metatheory

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

Decidability of Type Checking. The proof for decidability of type checking is by induction on the structure of e. The non-trivial case is for cast-terms with typing rules T_CastUp and T_CastDown. Both rules contain a premise that needs to judge if two types τ_1 and τ_2 follow the one-step reduction, i.e., if $\tau_1 \longrightarrow \tau_2$ holds. We show that 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. Such property is given by the following lemma:

Lemma 1 (Determinacy of One-step Weak-head Reduction). *If $e \longrightarrow e_1$ and $e \longrightarrow e_2$, then $e_1 \equiv e_2$.*

We use the notation $\equiv$ to denote the *alpha* equivalence of e_1 and e_2. Note that the presence of recursion does not affect this lemma: given a recursive term $\mu x : \tau.\ e$, by rule S_Mu, there always exists a unique term $e' \equiv e[x \mapsto \mu x : \tau.\ e]$ such that $\mu x : \tau.\ e \longrightarrow e'$. With this result, we show it is decidable to check whether the one-step relation $\tau_1 \longrightarrow \tau_2$ holds. We first reduce τ_1 by one step to obtain τ_1' (which is unique by Lemma 1), and compare if τ_1' and τ_2 are syntactically equal. Thus, we can further show type checking cast-terms is decidable.

For other forms of terms, the typing rules only contain typing judgments in the premises. Thus, type checking is decidable by the induction hypothesis and the following lemma which ensures the typing result is unique:

Lemma 2 (Uniqueness of Typing for λI_{w}). *If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 \equiv \tau_2$.*

Thus, we can conclude the decidability of type checking:

Theorem 1 (Decidability of Type Checking for λI_{w}). *Given a well-formed context Γ and a term e, it is decidable to determine if there exists τ such that $\Gamma \vdash e : \tau$.*

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

Type Safety. The proof of the type safety of λI_{w} is by showing subject reduction and progress theorems:

Theorem 2 (Subject Reduction of λI_{w}). *If $\Gamma \vdash e : \sigma$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : \sigma$.*

Theorem 3 (Progress of λI_{w}). *If $\varnothing \vdash e : \sigma$ then either e is a value v or there exists e' such that $e \longrightarrow e'$.*

The proof of subject reduction is straightforward by induction on the derivation of $e \longrightarrow e'$. Some cases need supporting lemmas: S_CastElim requires Lemma 1; S_Beta and S_Mu require the following substitution lemma:

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

The proof of progress is also standard by induction on $\varnothing \vdash e : \sigma$. Notice that $\mathsf{cast}_{\uparrow}\,[\tau]\,v$ is a value, while $\mathsf{cast}_{\downarrow}\,e_1$ is not: by rule S_CastDown, e_1 will be constantly reduced until it becomes a value that could only be in the form $\mathsf{cast}_{\uparrow}\,[\tau]\,v$ by typing rule T_CastDown. Then rule S_CastElim can be further applied and the evaluation does not get stuck. Another notable remark is that when proving the case for application $e_1\,e_2$, if e_1 is a value, it could only be a λ-term but not a $\mathsf{cast}_{\uparrow}$-term. Otherwise, suppose e_1 has the form $\mathsf{cast}_{\uparrow}\,[\Pi x : \tau_1.\ \tau_2]\,e_1'$. By inversion, we have $\varnothing \vdash e_1' : \tau_1'$ and $\Pi x : \tau_1.\ \tau_2 \longrightarrow \tau_1'$. But such τ_1' does not exist because $\Pi x : \tau_1.\ \tau_2$ is a value which is not reducible.

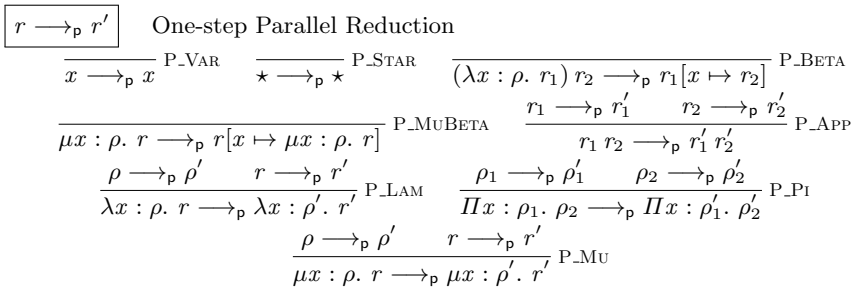

Fig. 4. One-step parallel reduction of erased terms

4 Iso-Types with Full Casts

In Section 3, casts use one-step *weak-head* reduction, which is also used by term evaluation and simplifies the design. To gain extra expressiveness, we take one step further to generalize casts with *full* reduction. In this section, we present a variant of λI called λI_{p}, where casts use *parallel reduction* for type conversion. Full specification and proofs can be found in the extended version [29].

4.1 Full Casts with Parallel Reduction

Using weak-head reduction in cast operators keeps the simplicity of the language design. However, it lacks the ability to do *full* type-level computation, because reduction cannot occur at certain positions of terms. For example, weak casts cannot convert the type *Vec* $(1+1)$ to *Vec* 2 since the desired reduction is at the non-head position. Thus, we generalize weak casts to *full* casts ($\mathsf{cast}_{\Uparrow}$ and $\mathsf{cast}_{\Downarrow}$) utilizing *one-step parallel reduction* ($\longrightarrow_{\mathsf{p}}$) for type conversion. Figure 4 shows the definition of $\longrightarrow_{\mathsf{p}}$. It allows to reduce terms at any position, e.g., non-head positions or inside binders $\lambda x : \star.\ 1+1 \longrightarrow_{\mathsf{p}} \lambda x : \star.\ 2$, thus enables full type-level computation for casts.

There are three remarks for parallel reduction worth mentioning. First, parallel reduction is defined up to *erasure*, a process that removes all casts from terms (see Figure 5). We use metavariable r and ρ to range over erased terms and types, respectively. The only syntactic change of erased terms is that there is no cast. The syntax is omitted here and can be found in the extended version [29]. It is feasible to define parallel reduction only for erased terms because casts in λI_{p} (also λI_{w}) are only used to ensure the decidability of type checking and have no effect on dynamic semantics, thus are *computationally irrelevant*.

Second, the definition of parallel reduction in Figure 4 is slightly different from the standard one for PTS [1]. It is *partially* parallel: rules P_Beta and P_MuBeta do not parallel reduce sub-terms but only do beta reduction and recursion unrolling, respectively. Such definition makes the decidability property (see Lemma 6) easier to prove than the conventional fully parallel version. It also requires fewer reduction steps than the non-parallel version, thus correspondingly needs fewer casts.

Erased Expressions r, ρ

Erasure

$$
\begin{array}{ll}
|x| & = x \\
|\star| & = \star \\
|e_1\ e_2| & = |e_1|\ |e_2| \\
|\lambda x : \tau.\ e| & = \lambda x : |\tau|.\ |e| \\
|\Pi x : \tau_1.\ \tau_2| & = \Pi x : |\tau_1|.\ |\tau_2| \\
|\mu x : \tau.\ e| & = \mu x : |\tau|.\ |e| \\
|\mathsf{cast}_{\Uparrow}\ [\tau]\ e| & = |e| \\
|\mathsf{cast}_{\Downarrow}\ [\tau]\ e| & = |e|
\end{array}
$$

Fig. 5. Erasure of casts

Erased Values

$u ::= \star \mid \lambda x : \rho.\ r \mid \Pi x : \rho_1.\ \rho_2$

Evaluation Rules

$$\frac{}{(\lambda x : \rho.\ r_1)\ r_2 \longrightarrow r_1[x \mapsto r_2]}$$

$$\frac{r_1 \longrightarrow r_1'}{r_1\ r_2 \longrightarrow r_1'\ r_2}$$

$$\frac{}{\mu x : \rho.\ r \longrightarrow r[x \mapsto \mu x : \rho.\ r]}$$

Fig. 6. Values and evaluation rules of erased terms

Expressions $e, \tau, \sigma ::= \cdots \mid \mathsf{cast}_{\Uparrow}\ [\tau]\ e \mid \mathsf{cast}_{\Downarrow}\ [\tau]\ e$

Values $v ::= \cdots \mid \mathsf{cast}_{\Uparrow}\ [\tau]\ v \mid \mathsf{cast}_{\Downarrow}\ [\tau]\ v$

Typing

$$\frac{\Gamma \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 : \star \qquad |\tau_1| \longrightarrow_{\mathsf{p}} |\tau_2|}{\Gamma \vdash \mathsf{cast}_{\Uparrow}\ [\tau_1]\ e : \tau_1}\ \textsc{TF\_CastUp}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_2 : \star \qquad |\tau_1| \longrightarrow_{\mathsf{p}} |\tau_2|}{\Gamma \vdash \mathsf{cast}_{\Downarrow}\ [\tau_2]\ e : \tau_2}\ \textsc{TF\_CastDown}$$

Fig. 7. Syntactic and typing changes of λI_{p}

Third, parallel reduction does *not* have the determinacy property like weak-head reduction (Lemma 1). For example, for term $(\lambda x : \star.\ 1 + 1)\ Int$, we can (parallel) reduce it to either $(\lambda x : \star.\ 2)\ Int$ by rule P_App and P_Lam, or $1 + 1$ by rule P_Beta. Thus, to ensure the decidability, we also need to add the type annotation for $\mathsf{cast}_{\Downarrow}$ operator to indicate what exact type we want to reduce to. Similar to $\mathsf{cast}_{\Uparrow}$, $\mathsf{cast}_{\Downarrow}\ [\tau]\ v$ is a value, which is different from the weak $\mathsf{cast}_{\downarrow}$-term.

Figure 7 shows the syntactic and typing changes of λI_{p}. Notice that in λI_{w}, reduction rules for type casting and term evaluation are the *same*, i.e., the weak-head call-by-name reduction. But in λI_{p}, parallel reduction is only used by casts. We define *weak-head* reduction ($\longrightarrow$) for term evaluation individually (see Figure 6). Note that the relation $\longrightarrow$ is defined only for *erased terms*, which is similar to the treatment of $\longrightarrow_{\mathsf{p}}$. We also define syntactic values for erased terms, ranged over by u (see Figure 6).

4.2 Metatheory

We show that the two key properties, type safety and decidability of type checking, still hold in λI_{p}.

Type Safety. Full casts are more expressive but also complicate the metatheory: term evaluation could get stuck by full casts. For example, the following term,

$$(\mathsf{cast}_{\Downarrow}\ [Int \to Int]\ (\lambda x : ((\lambda y : \star.\ y)\ Int).\ x))\ 3$$

cannot be further reduced because the head position is already a value but not a λ-term. Note that weak casts do not have such problem because only $\mathsf{cast}_{\uparrow}$ is annotated and not legal to have a Π-type in the annotation (see last paragraph of Section 3.5). To avoid getting stuck by full casts, one could introduce several *cast push rules* similar to System F_C [26]. For example, the stuck term above can be further evaluated by pushing $\mathsf{cast}_{\Downarrow}$ into the λ-term:

$$(\mathsf{cast}_{\Downarrow}\,[\mathit{Int} \to \mathit{Int}]\,(\lambda x : ((\lambda y : \star.\ y)\,\mathit{Int}).\ x))\,3 \longrightarrow (\lambda x : \mathit{Int}.\ x)\,3$$

However, adding "push rules" significantly complicates the reduction relations and metatheory. Instead, we adopt the erasure approach inspired by Zombie [24] and GURU [25] that removes all casts when proving the type safety. We define a type system for erased terms, called *erased system*. Its typing judgment is $\Delta \vdash r : \rho$ where Δ ranges over the erased context. Omitted typing rules are available in the extended version [29].

The erased system is basically calculus of constructions with recursion and "type-in-type". Thus, we follow the standard proof steps for PTS [5]. Notice that term evaluation uses the weak-head reduction $\longrightarrow$. We only need to prove subject reduction and progress theorems for $\longrightarrow$. But we generalize the result for subject reduction, which holds up to the parallel reduction $\longrightarrow_{\mathsf{p}}$.

Lemma 4 (Substitution of Erased System). *If $\Delta_1, x : \rho', \Delta_2 \vdash r_1 : \rho$ and $\Delta_1 \vdash r_2 : \rho'$, then $\Delta_1, \Delta_2[x \mapsto r_2] \vdash r_1[x \mapsto r_2] : \rho[x \mapsto r_2]$.*

Theorem 4 (Subject Reduction of Erased System). *If $\Delta \vdash r : \rho$ and $r \longrightarrow_{\mathsf{p}} r'$ then $\Delta \vdash r' : \rho$.*

Theorem 5 (Progress of Erased System). *If $\varnothing \vdash r : \rho$ then either r is a value u or there exists r' such that $r \longrightarrow r'$.*

Given that the erased system is type-safe, if we want to show the type-safety of the original system, it is sufficient to show the typing is preserved after erasure:

Lemma 5 (Soundness of Erasure). *If $\Gamma \vdash e : \tau$ then $|\Gamma| \vdash |e| : |\tau|$.*

Decidability of Type Checking. The proof of decidability of type checking λI_{p} is similar to λI_{w} in Section 3.5. The only difference is for cast rules TF_CASTUP and TF_CASTDOWN, which use parallel reduction $|\tau_1| \longrightarrow_{\mathsf{p}} |\tau_2|$ as a premise. We first show the decidability of parallel reduction:

Lemma 6 (Decidability of Parallel Reduction). *If $\Delta \vdash r_1 : \rho_1$ and $\Delta \vdash r_2 : \rho_2$, then whether $r_1 \longrightarrow_{\mathsf{p}} r_2$ holds is decidable.*

As $\mathsf{cast}_{\Uparrow}$ and $\mathsf{cast}_{\Downarrow}$ are annotated, both τ_1 and τ_2 can be determined and the well-typedness is checked in the original system. By Lemma 5, the erased terms keeps the well-typedness. Thus, by Lemma 6, it is decidable to check if $|\tau_1| \longrightarrow_{\mathsf{p}} |\tau_2|$. We conclude the decidability of type checking by following lemmas:

Lemma 7 (Uniqueness of Typing for λI_{p}). *If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 \equiv \tau_2$.*

Theorem 6 (Decidability of Type Checking for λI_{p}). *Given a well-formed context Γ and a term e, it is decidable to determine if there exists τ such that $\Gamma \vdash e : \tau$.*

5 Application of Iso-Types

λI is a simple core calculus, but expressive enough to encode useful language constructs. We have implemented a simple functional language **Fun** to show how features of modern functional languages can be encoded in λI. We focus on common features available in traditional functional languages and some interesting type-level features, but not the *full power* of dependent types. Supported features include algebraic datatypes, records, higher-kinded types, kind polymorphism [30] and datatype promotion [30].

Due to lack of space, many examples illustrating the various language features supported in **Fun** are provided only in the extended version [29]. Here we show the essential idea of how to exploit iso-types to encode language constructs.

Encoding Parametrized Algebraic Datatypes with Weak Casts. We give an example of encoding parametrized algebraic datatypes in λI_{w} via μ-operator and *weak* casts. Importantly we should note that having iso-recursive types alone (and alpha equality) would be insufficient to encode parametrized types: the generalization afforded by iso-types is needed here. In **Fun** we can define *polymorphic list* as:

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

This **Fun** definition is translated into λI_{w} using a Scott encoding [18] of datatypes:

$$\begin{aligned}
\mathit{List} &= \mu L : \star \to \star.\, \lambda a : \star.\, \Pi b : \star.\, b \to (a \to L\ a \to b) \to b \\
\mathit{Nil} &= \lambda a : \star.\, \mathsf{cast}^2_{\uparrow}\ [\mathit{List}\ a]\ (\lambda b : \star.\, \lambda n : b.\, \lambda c : (a \to \mathit{List}\ a \to b).\, n) \\
\mathit{Cons} &= \lambda a : \star.\, \lambda x : a.\, \lambda (xs : \mathit{List}\ a). \\
&\qquad \mathsf{cast}^2_{\uparrow}\ [\mathit{List}\ a]\ (\lambda b : \star.\, \lambda n : b.\, \lambda c : (a \to \mathit{List}\ a \to b).\, c\ x\ xs)
\end{aligned}$$

The type constructor *List* is encoded as a recursive type. The body is a *type-level function* that takes a type parameter a and returns a dependent function type, i.e., Π-type. The body of Π-type is universally quantified by a type parameter b, which represents the result type instantiated during pattern matching. Following are the types corresponding to data constructors: b for *Nil*, and $a \to L\ a \to b$ for *Cons*, and the result type b at the end. The data constructors *Nil* and *Cons* are encoded as functions. Each of them selects a different function from the parameters (n and c). This provides branching in the process flow, based on the constructors. Note that $\mathsf{cast}_{\uparrow}$ is used twice here (written as $\mathsf{cast}^2_{\uparrow}$): one for *one-step expansion* from τ to $(\lambda a : \star.\, \tau)\, a$ and the other for *folding the recursive type* from $(\lambda a : \star.\, \tau)\, a$ to $\mathit{List}\ a$, where τ is the type of $\mathsf{cast}^2_{\uparrow}$ body.

We have two notable remarks from the example above. First, iso-types are critical for the encoding and cannot be replaced by iso-recursive types. Since type constructors are parameterized, not only folding/unfolding recursive types, but also type-level reduction/expansion is required, which is only possible with casts. Second, though weak casts are not as powerful as full casts, they are capable of encoding many useful constructs, such as algebraic datatypes and records [29]. Nevertheless full-reduction casts enable other important applications. Some applications of full casts are discussed in the extended version [29].

6 Related Work

Core calculus for functional languages. Girard's System F_ω [14] is a typed lambda calculus with higher-kinded polymorphism. For the well-formedness of type expressions, an extra level of *kinds* is added to the system. In comparison, because of unified syntax, λI is considerably simpler than System F_ω, both in terms of language constructs and complexity of proofs. As for type-level computation, System F_ω differs from λI in that it uses a conversion rule, while λI uses explicit casts. The current core language for GHC Haskell, System F_C [26] is a significant extension of System F_ω, which supports GADTs [20], functional dependencies [15], type families [13], and kind equality [28]. These features use a non-trivial form of type equality, which is currently missing from λI. On the other hand, λI uses unified syntax and has only 8 language constructs, whereas System F_C uses multiple levels of syntax and currently has over 30 language constructs, making it significantly more complex. One direction of our future work is to investigate the addition of such forms of non-trivial type-equality.

Unified syntax with decidable type-checking. Pure Type Systems [4] show how a whole family of type systems can be implemented using just a single syntactic form. PTSs are an obvious source of inspiration for our work. Although this paper presents a specific system based on λC, it should be easy to generalize λI in the same way as PTSs and further show the applicability of our ideas to other systems. An early attempt of using a PTS-like syntax for an intermediate language for functional programming was Henk [21]. The Henk proposal was to use the *lambda cube* as a typed intermediate language, unifying all three levels. However the authors have not studied the addition of general recursion nor full dependent types.

Zombie [8] is a dependently typed language using a single syntactic category. It is composed of two fragments: a logical fragment where every expression is known to terminate, and a programmatic fragment that allows general recursion. Though Zombie has one syntactic category, it is still fairly complicated (with around 24 language constructs) as it tries to be both consistent as a logic and pragmatic as a programming language. Even if one is only interested in modeling a programmatic fragment, additional mechanisms are required to ensure the validity of proofs, e.g., call-by-value semantics and value restriction [23, 24]. In contrast to Zombie, λI takes another point of the design space, giving up logical consistency and reasoning about proofs for simplicity in the language design.

Unified syntax with general recursion and undecidable type checking. Cayenne [3] integrates the full power of dependent types with general recursion, which bears some similarities with λI. It uses one syntactic form for both terms and types, allows arbitrary computation at type level and is logically inconsistent because of allowing unrestricted recursion. However, the most crucial difference from λI is that type checking in Cayenne is *undecidable*. From a pragmatic point of view, this design choice simplifies the implementation, but the desirable property of decidable type checking is lost. Cardelli's Type:Type language [7] also features general recursion to implement equi-recursive types. Recursion and recursive types are unified in a single construct. However, both equi-recursive types and the Type:Type axiom make the type system undecidable. $\Pi\Sigma$ [2] is another example of a language that uses one recursion mechanism for both types and functions. The type-level recursion is controlled by lifted types and boxes since definitions are not unfolded inside boxes. However, $\Pi\Sigma$ does not have decidable type checking due to the "type-in-type" axiom. And its metatheory is not formally developed.

Casts for managed type-level computation. Type-level computation in λI is controlled by explicit casts. Several studies [12,17,23–26] also attempt to use explicit casts for managed type-level computation. However, casts in those approaches are not inspired by iso-recursive types. Instead they require *equality proof terms*, while casts in λI do not. The need for equality proof terms complicates the language design because: 1) building equality proofs requires various other language constructs, adding to the complexity of the language design and metatheory; 2) It is desirable to ensure that the equality proofs are valid. Otherwise, one can easily build bogus equality proofs with non-termination, which could endanger type safety. Guru [25] and Sep3 [17] make *syntactic separation* between proofs and programs to prevent certain programmatic terms turning into invalid proofs. The programmatic part of Zombie [23,24], which has no such separation, employs *value restriction* that restricts proofs to be syntactic values to avoid non-terminating terms. PTS with convertibility proofs (PTS_f) [12] extends PTS by replacing the implicit conversion rule with explicit conversion proofs embedded into terms. However, it requires many language constructs to build equality proofs; and it does not allow general recursion, thus does not need to deal with problem 2). Our treatment of full casts in λI_{p}, using a separate erased system for developing metatheory, is similar to the approach of Zombie or Guru which uses an unannotated system.

Restricted recursion with termination checking. As proof assistants, dependently typed languages such as Coq [9] and Adga [19] are conservative as to what kind of computation is allowed. They require all programs to terminate by means of a termination checker, ensuring recursive calls are decreasing. Decidable type checking and logical consistency are preserved. But the conservative, syntactic criteria is insufficient to support a variety of important programming paradigms. Agda offers an option to disable the termination checker to allow writing arbitrary functions. However, this may endanger both decidable type checking and

logical consistency. Idris [6] is a dependently typed language that allows writing unrestricted functions. However, to achieve decidable type checking, it also requires termination checker to ensure only terminating functions are evaluated by the type checker. While logical consistency is an appealing property, it is not a goal of λI. Instead λI aims at retaining (term-level) general recursion as found in languages like Haskell or ML, while benefiting from a unified syntax to simplify the implementation of the core language.

7 Conclusion

This work proposes λI: a minimal dependently typed core language that allows the same syntax for terms and types, supports type-level computation, and preserves decidable type checking under the presence of general recursion. The key idea is to control type-level computation using iso-types via casts. Because each cast can only account for one-step of type-level computation, type checking becomes decidable without requiring strong normalization of the calculus. At the same time one-step casts together with recursion provide a generalization of iso-recursive types. Two variants of λI show trade-offs of employing different reduction strategies in casts. In future work, we hope to investigate surface language mechanisms, such as type families in Haskell, to express intensive type-level computation in a more convenient way.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. This work has been sponsored by the Hong Kong Research Grant Council Early Career Scheme project number 27200514.

References

1. Adams, R.: Pure type systems with judgemental equality. Journal of Functional Programming 16(02), 219–246 (2006)
2. Altenkirch, T., Danielsson, N.A., Löh, A., Oury, N.: $\Pi\Sigma$: Dependent types without the sugar. In: Functional and Logic Programming, pp. 40–55. Springer (2010)
3. Augustsson, L.: Cayenne — a language with dependent types. In: ICFP '98. pp. 239–250 (1998)
4. Barendregt, H.: Introduction to generalized type systems. Journal of Functional Programming 1(2), 125–154 (1991)
5. Barendregt, H.: Lambda calculi with types. In: Handbook of Logic in Computer Science. vol. 2, pp. 117–309 (1992)
6. Brady, E.: IDRIS—systems programming meets full dependent types. In: PLPV '11. pp. 43–54 (2011)
7. Cardelli, L.: A Polymorphic lambda-calculus with Type: Type. Digital Systems Research Center (1986)
8. Casinghino, C., Sjöberg, V., Weirich, S.: Combining proofs and programs in a dependently typed language. In: POPL '14. pp. 33–45 (2014)
9. Coq development team: The coq proof assistant. `http://coq.inria.fr/`

10. Coquand, T., Huet, G.: The calculus of constructions. Information and Computation 76, 95–120 (1988)
11. Crary, K., Harper, R., Puri, S.: What is a recursive module? In: PLDI '99. pp. 50–63 (1999)
12. van Doorn, F., Geuvers, H., Wiedijk, F.: Explicit convertibility proofs in pure type systems. In: LFMTP '13. pp. 25–36 (2013)
13. Eisenberg, R.A., Vytiniotis, D., Peyton Jones, S., Weirich, S.: Closed type families with overlapping equations. In: POPL '14 (2014)
14. Girard, J.Y.: Interprtation fonctionnelle et limination des coupures de l'arithmtique d'ordre suprieur. Ph.D. thesis, Universit Paris VII (1972)
15. Jones, M.P.: Type Classes with Functional Dependencies. Proceedings of the 9th European Symposium on Programming Languages and Systems (March) (2000)
16. Jutting, L.: Typing in pure type systems. Information and Computation 105(1), 30–41 (1993)
17. Kimmell, G., Stump, A., III, H.D.E., Fu, P., Sheard, T., Weirich, S., Casinghino, C., Sjöberg, V., Collins, N., Ahn, K.Y.: Equational reasoning about programs with general recursion and call-by-value semantics. In: PLPV '12. pp. 15–26 (2012)
18. Mogensen, T.A.: Theoretical pearls: Efficient self-interpretation in lambda calculus. Journal of Functional Programming 2(3), 345–364 (1992)
19. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)
20. Peyton Jones, S., Washburn, G., Weirich, S.: Wobbly types: type inference for generalised algebraic data types. Tech. Rep. MS-CIS-05-26, University of Pennsylvania (Jul 2004)
21. Peyton Jones, S., Meijer, E.: Henk: a Typed Intermediate Language. In: Types in Compilation Workshop (1997)
22. Pierce, B.C.: Types and programming languages. MIT press (2002)
23. Sjöberg, V., Casinghino, C., Ahn, K.Y., Collins, N., III, H.D.E., Fu, P., Kimmell, G., Sheard, T., Stump, A., Weirich, S.: Irrelevance, heterogenous equality, and call-by-value dependent type systems. In: MSFP '12. pp. 112–162 (2012)
24. Sjöberg, V., Weirich, S.: Programming up to congruence. In: POPL '15. pp. 369–382 (2015)
25. Stump, A., Deters, M., Petcher, A., Schiller, T., Simpson, T.: Verified programming in guru. In: PLPV '09. pp. 49–58 (2008)
26. Sulzmann, M., Chakravarty, M.M.T., Jones, S.P., Donnelly, K.: System f with type equality coercions. In: TLDI '07. pp. 53–66 (2007)
27. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: ICFP '11. pp. 266–278 (2011)
28. Weirich, S., Hsu, J., Eisenberg, R.A.: System fc with explicit kind equality. In: ICFP '13. pp. 275–286 (2013)
29. Yang, Y., Bi, X., Oliveira, B.C.d.S.: Unified syntax with iso-types. Extended version available from `https://bitbucket.org/ypyang/aplas16` (2016)
30. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving Haskell a Promotion. In: TLDI '12. pp. 53–66 (2012)