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, since properties such as decidable type-checking are easily lost.

This paper presents a family of calculi called pure iso-type systems (PITS), which employs unified syntax, supports general recursion and preserves decidable type-checking. PITS is comparable in simplicity to pure type systems (PTSs), and is useful to serve as a foundation for functional languages that stand in-between traditional ML-like languages and fully-blown dependently typed languages. In PITS recursion and recursive types are completely unrestricted and type equality is simply based on alpha-equality, just like traditional ML-style languages. However, like most dependently typed languages, PITS uses unified syntax, naturally supporting many advanced type system features. Instead of implicit type conversion, PITS provides a generalization of iso-recursive types called iso-types. Iso-types replace the conversion rule typically used in dependently typed calculus, and make every type-level computation explicit via cast operators. Iso-types avoid the complexity of explicit equality proofs employed in other approaches with casts. We study three variants of PITS that differ on the reduction strategy employed by the cast operators: call-by-name, call-by-value and parallel reduction. One key finding is that while using call-by-value or call-by-name reduction in casts loses some expressive power, it allows those variants of PITS to have simple and direct operational semantics and proofs. In contrast the variant of PITS with parallel reduction retains the expressive power of PTS conversion, at the cost of a more complex metatheory.

1 Introduction

Dependent types are gaining popularity in functional programming language design and research in recent years (Augustsson, 1998; Altenkirch et al., 2010; Sjöberg et al., 2012; Stump et al., 2008; Weirich et al., 2013; Gundry, 2013; Casinghino et al., 2014; Sjöberg & Weirich, 2015; Sjöberg, 2015; Eisenberg, 2016; Weirich et al., 2017). Several new full-spectrum dependently languages, including Agda (Norell, 2007) or Idris (Brady, 2011), are emerging. These languages are designed to be both programming languages and proof assistants. As such they support different forms of assurances, such as strong normalization and logical consistency, not typically present in traditional programming languages such as Haskell and ML. 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. Incorporating recursion and recursive types in such languages needs careful checks (e.g. termination and positivity) for not breaking the logical consistency and strong normalization. In contrast Haskell or ML programmers can write their programs much more freely and can support general recursion and recursive types easily, since they do not need to retain strong normalization and logical consistency. Thus the design space between ML-style languages and full-blown dependently typed languages like Agda or Idris is very large.

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 (Eisenberg et al., 2016) (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 do not have strong normalization or logical consistency and have only restricted forms of type-level computation. Not surprisingly, researchers have in the past considered options for implementing functional languages based on unified syntax (Augustsson, 1998; Cardelli, 1986; Peyton Jones & Meijer, 1997), by using some variant of pure type systems (PTSs) (Barendregt, 1992) (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 (Augustsson, 1998; Cardelli, 1986) 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.

For 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 often desirable in a traditional programming language design. Decidable type checking implies that an algorithm that checks whether an expression is typeable or not exists. It is natural to ask the question of whether it is possible to come up with a simpler language design for unified syntax and general recursion, if all we wish to preserve is decidable type-checking. Surprisingly, there are not so many designs that attempt to support unified syntax and general recursion with decidable type-checking. Notable exceptions are Dependently Typed Haskell (Weirich et al., 2017) (DTH) and PTSs with explicit
convertibility proofs (PTS) (van Doorn et al., 2013). However these works are still quite involved due to the need of a separate language for building equality proof terms.

This paper presents a family of calculi called pure iso-type systems (PITS), which employs unified syntax, supports general recursion and preserves decidable type-checking. PITS is comparable in simplicity to PTSs 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. Such design choice is useful to serve as a foundation for functional languages that stand in-between traditional ML-like languages and fully-blown dependently typed languages. In PITS, recursion and recursive types are completely unrestricted and type equality is simply based on alpha-equality, just like traditional ML-style languages. However, like most dependently typed languages, PITS uses unified syntax, naturally supporting many advanced type system features (such as higher-kindned types (Girard, 1972), or kind polymorphism (Yorgey et al., 2012)).

PITS design means that instead of an implicit type conversion (employed by PTS), PITS provides a generalization of iso-recursive types (Crary et al., 1999; Pierce, 2002) called iso-types. Iso-types replace the conversion rule typically used in dependently typed calculus, and make every type-level computation explicit via cast operators. While such explicit casts would make many forms of dependently typed programming inconvenient, they are sufficient to express the kind of type-level computation required by more traditional language designs. We study three variants of PITS that differ on the choice of the reduction strategy used by cast operators: call-by-name, call-by-value or parallel reduction. The different variants give different trade-offs in terms of simplicity and expressiveness, which we discuss thoroughly in the paper.

One key finding is that while using call-by-value or call-by-name reduction in casts loses some expressive power for type-level computation, it allows those variants of PITS to have simple and direct operational semantics and proofs. In contrast the variant of PITS with parallel reduction retains the expressive power of PTS conversion, at the cost of a more complex metatheory where type-safety proofs must be shown indirectly by showing soundness/completeness to another variant of PTS. A detailed discussion of the trade-offs between the variants of PITS, as well as PTS, is given in Section 7.1.

To demonstrate the application of PITS, we build a simple surface language Fun that extends PITS with algebraic datatypes by using a Scott encoding of datatypes (Mogensen, 1992). We also implement prototype interpreter and compiler for Fun, which can run all examples shown in this paper.

In summary, the contributions of this work are:

- **PITS**: A variant of PTS with general recursion, unified syntax and decidable type-checking.
- **Iso-types**: A generalization of iso-recursive types, which makes all type-level computation steps explicit via casts operators. The combination of casts and recursion subsumes iso-recursive types.
Y. Yang and B. Oliveira

- **Studying three different reduction strategies:** We study PITS with three different reduction strategies: call-by-name PITS, call-by-value PITS and PITS with parallel reduction at the type-level. One key conclusion is that call-by-name and call-by-value PITS have very simple and direct operational semantics and proofs. The added expressive power brought by parallel reduction complicates the semantics and metatheory, and requires a more complex formalization.

- **Mechanized proofs:** All proofs of PITS in the paper are machine-checked in Coq theorem prover (The Coq development team, 2016) and available online.

- **Prototype implementation:** Prototype interpreter and compiler for Fun, a simple language based on PITS extended with algebraic datatypes, are implemented and available online.

This article is a significantly revised and expanded version of a conference paper (Yang et al., 2016). There are three main novelties with respect to the conference version. Firstly, while the conference version of the paper only deals with a calculus of constructions (Coquand & Huet, 1988) style calculus (with the “type-in-type” axiom), PITS is generalized to deal with a whole family of calculi in the PTS tradition. Secondly, the study of call-by-value is new. Finally, for the third variant with parallel reduction, we prove a completeness theorem that demonstrates that our parallel reduction relation subsumes full beta-reduction used in PTS. All related online materials are available from:

https://bitbucket.org/ypyang/pits-jfp

2 Motivation and Overview

In this section, we motivate the design and informally introduce the main features of PITS. In particular, we show how iso-types can be used to allow general recursion, instead of the typical conversion rule in PTS. We also give examples to illustrate how features of modern functional languages can be encoded in PITS. The formal details of PITS are presented in Sections 4, 5 and 6.

2.1 Implicit Type Conversion in Pure Type Systems

Pure Type Systems (PTS) (Barendregt, 1991) are a generic framework to study a family of type systems, including the Simply Typed Lambda Calculus (STLC), System F (Girard, 1972) and the Calculus of Constructions (Coquand & Huet, 1988). A PTS is determined by a triple \((\text{Sorts}, \mathcal{A}, \mathcal{R})\) (Barendregt, 1991) where \(\mathcal{A} \subseteq \text{Sorts} \times \text{Sorts}\) is a set of Axioms to type sorts and \(\mathcal{R} \subseteq \text{Sorts} \times \text{Sorts} \times \text{Sorts}\) is a set of Rules to type II-types. By specifying the PTS triple, we can obtain specific type systems. For example, given \(\text{Sort} = \{*, \Box\}\) and \(\mathcal{A} = \{(\ast, \Box)\}\), if \(\mathcal{R} = \{(\ast, \ast, \ast)\}\), we obtain STLC; if \(\mathcal{R} = \{(\ast, \ast, \ast), (\Box, \ast, \ast)\}\), we obtain System F.

The typing rules for PTS contain a conversion rule:

\[
\Gamma \vdash e : A \quad A =_\beta B \\
\Gamma \vdash e : B
\]
Pure Iso-Type Systems

This rule allows one to derive \( e : B \) from the derivation of \( e : A \) with the beta equality of \( A \) and \( B \). This rule is important to automatically allow terms with beta equivalent types to be considered type-compatible. For example, consider the following identity function:

\[
f = \lambda y : (\lambda x : *. x) \text{Int. } y
\]

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

Decidability of Type Checking and Strong Normalization. While the conversion rule in PTS brings a lot of convenience, an unfortunate consequence is that it couples decidability of type checking with strong normalization of the calculus (van Benthem Jutting, 1993). Therefore adding general recursion to PTS 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 \( \text{loop} \), where \( \text{loop} \) stands for any diverging computation. If we type check \((\lambda x : \text{Int. } x) z\) under the normal typing rules of PTS, the type checker would get stuck as it tries to normalize and compare two terms: \( \text{Int} \) and \( \text{loop} \), where normalizing the latter is non-terminating.

2.2 Newtypes: A Mechanism for Explicit Type Conversion in Haskell

Early designs of functional languages such as Haskell deliberately forbid implicit type conversions. However, although not widely appreciated, since the very beginning Haskell (and other functional languages) has supported explicit type-conversions via algebraic datatypes, or their simpler sibling newtypes. In essence, encapsulated behind algebraic datatypes and newtypes is a language mechanism that supports explicit type conversions. Such language mechanism is closely related to iso-recursive types, but also allows for computations that are not recursive.

In early versions of Haskell, such as Haskell 98, there is no real type-level computation as in dependently-typed languages such as Coq (The Coq development team, 2016). In particular, there are no computational type-level lambdas. Indeed Haskell forbids type-level lambdas to avoid higher-order unification that is required in dependently typed languages such as Coq or Agda (Jones, 1993). While modern Haskell is half the way in evolving into a dependently typed language, it still does not support type-level lambdas.

Despite the absence of type-level lambdas, it is still possible to express type-level functions via newtype or data constructs. For example, the type \( \text{Id} \) defined by

\[
\text{newtype Id } a = \text{MkId } \{ \text{runId} :: a \}
\]

can be viewed as a type-level identity function and \( \text{Id } a \) is isomorphic to type \( a \). To convert the type back and forth between \( a \) and \( \text{Id } a \), one needs to explicitly use the constructor \( \text{MkId} \) or the destructor \( \text{runId} \).
Iso-recursive Types in Haskell. More generally, it is possible to use newtypes to model recursive types in Haskell as well:

\begin{verbatim}
newtype Fix f = Fold { unfold :: f (Fix f) }
\end{verbatim}

The type \texttt{Fix} is a type-level fixpoint. Its constructor \texttt{Fold} forms a recursive type from function \texttt{f} and destructor \texttt{unfold} forces the computation to obtain the unrolling \texttt{f (Fix f)}. Such treatment of recursive types is well-studied, and it is essentially a form of iso-recursive types.

While the explicit type level computations enabled by newtypes or algebraic datatypes are very simple, they are essential for many of the characterizing programming styles employed in Haskell. For example without such simple explicit type conversions it would not be possible to have monadic programming (Wadler, 1995), or modular interpreters enabled by approaches such as Datatypes `a la Carte (Swierstra, 2008).

2.3 Iso-types: Explicit Type Conversion in PITS

The main source of inspiration for the design of PITS comes from iso-recursive types (Crary et al., 1999; Pierce, 2002). Iso-types offer an alternative to the conversion rule of PTS, making it explicit as to when and where to convert one type to another. Type conversions are explicitly controlled by two language constructs: \texttt{cast} (one-step reduction) and \texttt{cast} (one-step expansion). In PITS, 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 programming 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.

We view the design of traditional dependently typed calculi, such as PTS, as analogous to systems with equi-recursive types. In PTS, it is the conversion rule that allows type-level computation to by implicitly triggered. However, the proof of decidability of type checking for PTS is non-trivial, as it depends on strong normalization (van Benthem Jutting, 1993). Moreover decidability is lost when adding general recursion. In contrast, the cast operators in PITS have to be used to explicitly trigger each step of type-level computation, but it is easy to ensure decidable type-checking, even in the presence of general recursion. Another potential benefit of iso-types is that the problem of type-inference may be significantly simpler in a language with iso-types than in a language with a conversion rule. Jones’ work on type-inference for higher-kinded types in Haskell (Jones, 1993) seems to back up this idea. We leave for future work exploring type-inference and unification for dependently typed systems with iso-types.

PITS iso-types also have strong similarities with the explicit conversion mechanism found in languages like Haskell. Differently from Haskell (and similarly to iso-recursive
types) iso-types are purely structural, while Haskell datatypes and newtypes are nominal. Because iso-types are structural they can be directly represented with type-level lambdas (and other constructs). Moreover, since in PITS type equality is just alpha-equality, type-level lambdas are not problematic, since they do not trigger computation during type-checking.

Reduction. The cast₁ 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 cast₁, assume an identity function \( g = \lambda y : \text{Int}. y \) and a term \( e : (\lambda x : \ast. x) \text{Int} \). In contrast to PTS, we cannot directly apply \( g \) to \( e \) in PITS since the type of \( e \) \((\lambda x : \ast. x) \text{Int}\) is not syntactically equal to \( \text{Int} \). However, note that the reduction relation \((\lambda x : \ast. x) \text{Int} \leftrightarrow \text{Int}\) holds. Therefore, we can use cast₁ for the explicit (type-level) reduction:

\[
\text{cast}_1 e : \text{Int}
\]

Then the application \( g(\text{cast}_1 e) \) type checks.

Expansion. The dual operation of cast₁ is cast", 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 cast", let us revisit the example from Section 2.1:

\[
f = \lambda y : (\lambda x : \ast. x) \text{Int}. y
\]

We cannot apply \( f \) to 3 without the conversion rule. Instead, we can use cast" to expand the type of 3:

\[
(\text{cast}_2 [(\lambda x : \ast. x) \text{Int}] 3) : (\lambda x : \ast. x) \text{Int}
\]

Thus, the application \( f (\text{cast}_2 [(\lambda x : \ast. x) \text{Int}] 3) \) becomes well-typed. Intuitively, cast₂ performs expansion, as the type of 3 is \( \text{Int} \), and \((\lambda x : \ast. x) \text{Int}\) is the expansion of \( \text{Int} \) witnessed by \((\lambda x : \ast. x) \text{Int} \leftrightarrow \text{Int}\). Notice that for cast₂ 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 operators 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 \( e : (\lambda x : \ast. \lambda y : \ast. x) \text{IntBool} \). Given \( g = \lambda y : \text{Int}. y \), the expression \( g(\text{cast}_1 e) \) is ill-typed because \( \text{cast}_1 e \) has type \((\lambda y : \ast. \text{Int}) \text{Bool}\), which is not syntactically equal to \( \text{Int} \). Thus, we need another \( \text{cast}_1 \):

\[
\text{cast}_1 \ (\text{cast}_1 e) : \text{Int}
\]

to further reduce the type and allow the program \( g(\text{cast}_1 (\text{cast}_1 e)) \) to type check.

Analogy to Newtypes in Haskell. By using cast operators, we can model the Haskell example shown in Section 2.2:

\[
\text{Id} = \lambda x : \ast. x
\]
\[
\text{cast}_1 [\text{Id } a] : a \rightarrow \text{Id } a
\]
\[
\text{cast}_1 : \text{Id } a \rightarrow a
\]
cast\dagger and \texttt{cast}_\$ are analogous to the datatype constructor \texttt{MkId} and destructor \texttt{runId}, respectively. The difference is that PITS directly supports type-level lambdas without the need of using \texttt{newtype}, though PITS only has alpha equality without implicit beta conversion. In some sense, type-level lambdas in PITS are \textit{non-computational} as \texttt{newtype}-style “type functions” in Haskell. Nevertheless, we can still trigger type-level computation by casts, similarly to \texttt{newtype} constructors and destructors in Haskell.

**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 \((\lambda x : \text{Int}.) z\). This is because in PITS, the type checker only performs a syntactic comparison between \texttt{Int} and loop, instead of beta equality. Thus, it rejects the above application as ill-typed. By using explicit casts, the type checker does not need to normalize types and it becomes decidable.

**Variants of Casts.** A reduction relation is used in cast operators to convert types. We study three possible reduction relations: \textit{call-by-name} reduction, \textit{call-by-value} reduction and \textit{full} reduction. Call-by-name and call-by-value reduction cannot reduce sub-terms at certain positions (e.g., inside \( \lambda \) or \( \Pi \) binders), while full reduction can reduce sub-terms at any position. We also create three variants of PITS for each variant of casts. Specifically, full PITS uses a \textit{decidable parallel reduction} relation with full cast operators \texttt{cast}_\dagger and \texttt{cast}_\$). All variants reflect the idea of iso-types, but have trade-offs between simplicity and expressiveness: call-by-name and call-by-value PITS use the same reduction relation for both casts and evaluation to keep the system and metatheory simple, but lose some expressiveness, e.g. cannot convert \( \lambda x : \text{Int}. (1 + 1) \) to \( \lambda x : \text{Int}. 2 \). Full PITS is more expressive but results in a more complicated metatheory (see Section 6). Note that when generally referring to PITS, we do not specify the reduction strategy, which could be any variant.

### 2.4 General Recursion

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

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

\[
\text{fact} = \mu f : \text{Int} \to \text{Int}. \lambda x : \text{Int}. \text{if } x == 0 \text{ then } 1 \text{ else } x \times f (x - 1)
\]

We treat the \( \mu \) operator as a \textit{fixpoint}, which evaluates \( \mu x : A. e \) to its recursive unfolding \( e[x \mapsto \mu x : A. e] \). Term-level recursion in PITS works as in any standard functional language, e.g., \texttt{fact 3} produces 6 as expected (see Section 4.4).
Recursive Types. The same $\mu$ primitive is used at the type level to represent iso-recursive types (Crary et al., 1999). 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. This is also similar to the previous Haskell example of (iso-)recursive types with Fold and unfold (see Section 2.2). In call-by-name PITS, such isomorphism is witnessed by $\text{cast}_\uparrow$ and $\text{cast}_\downarrow$. In fact, $\text{cast}_\uparrow$ and $\text{cast}_\downarrow$ 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 3.

To demonstrate the use of casts with recursive types, we show the formation of the “hungry” type (Pierce, 2002) $H = \mu x : \star. \text{Int} \rightarrow x$. A term $z$ of type $H$ will accept one more integer every time when it is unfolded by a $\text{cast}_\downarrow$:

$$
\begin{align*}
\text{cast}_\downarrow (z) & : H \\
\text{cast}_\downarrow (\text{cast}_\uparrow (z)) & : H \\
\text{cast}_\downarrow (\text{cast}_\downarrow (\ldots (\text{cast}_\downarrow (z) \ldots) \ldots)) & : H
\end{align*}
$$

3 Applications

PITS is a simple core calculus, but expressive enough to encode useful language constructs. In order to show how features of modern functional languages can be encoded in PITS, we implemented a simple functional language Fun, a thin layer that is desugared to a specific PITS with only a single sort $\star$ and “type-in-type” axiom. Thus, Fun is not logically consistent due to the “type-in-type” axiom. We focus on common features available in traditional functional languages and some interesting type-level features, but not the full power of dependent types. In this section, we briefly introduce the implementation of Fun and present examples.

Encodable Features. Table 1 shows a summary of encodable features in Fun, including algebraic datatypes (Section 3.1), higher-kindred types (Section 3.2), datatype promotion (Section 3.2), high-order abstract syntax (Section 3.2) and object encodings (Section 3.3). The encoding of algebraic datatypes in Fun uses Scott encodings (Mogensen, 1992). The encoding itself uses casts, but the use of casts is completely transparent to programmers. In other words the elaboration process inserts the casts automatically. Thus in all the examples in 3.2 the casts are inferred automatically. We illustrate how to encode a simple form of objects in Section 3.3, but since we did not develop a source syntax for objects the encoding is manual and requires programmers to use casts explicitly. All the examples (except the last one) work in the 3 variants of PITS. We also discuss one final example on dependently typed vectors (Section 3.4) that only works with parallel reduction. Vectors are an example of what would be called a GADT (Xi et al., 2003; Cheney & Hinze, 2003; Peyton Jones et al., 2004) in Haskell terminology. All examples can run in the prototype interpreter and compiler.

3.1 Fun Implementation

Fun uses PITS as its core language and provides surface language constructs for algebraic datatypes and pattern matching. The core language is a specific call-by-name PITS
Table 1. Encodable Features of Fun

<table>
<thead>
<tr>
<th>Encodable Features</th>
<th>Section</th>
<th>Encoding Method</th>
<th>Casts Inferred</th>
<th>Full Casts Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algebraic Datatypes</td>
<td>3.1</td>
<td>Scott Encoding</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Higher-kinded Types</td>
<td>3.2</td>
<td>Directly by Datatypes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Datatype Promotion</td>
<td>3.2</td>
<td>Directly by Datatypes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Higher-order Abstract Syntax</td>
<td>3.2</td>
<td>Directly by Datatypes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Object Encodings</td>
<td>3.3</td>
<td>Existential Types</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Vectors and GADTs</td>
<td>3.4</td>
<td>Leibniz Equality</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

called $\lambda I$ (Yang et al., 2016). $\lambda I$ specifies the PITS triple (see Section 2.1) as $\text{Sort} = \{\ast\}$, $\mathcal{A} = \{\ast, \ast\}$ and $\mathcal{R} = \{\ast, \ast, \ast\}$. Algebraic datatypes and pattern matching in Fun are implemented using Scott encodings (Mogensen, 1992), which can be later desugared into PITS ($\lambda I$) terms. For demonstration, we implemented a prototype interpreter and compiler for Fun, both written in GHC Haskell (Marlow, 2010). Fun terms are firstly desugared into $\lambda I$ terms and then type-checked using PITS typing rules. The type-checked $\lambda I$ terms can be evaluated directly by the interpreter or compiled to JavaScript or Haskell code.

Inferring Casts. As a surface language, explicit casts are mostly intended to be generated by the compiler, but not by the programmers. Indeed, some casts in Fun are automatically inferred by elaboration rules. In particular the elaboration of algebraic datatypes and pattern matching infers casts. Since the focus of this work is to illustrate the design space of the core language with casts we do have not yet explored the addition of surface-level mechanisms for doing more advanced type-level computation built on top of casts. Languages like Haskell have a similar design to Fun with a very explicit core language with casts. In Haskell cast inference is taken to an extreme. Many surface features in Haskell, e.g., type families (Schrijvers et al., 2008) and associated types (Chakravarty et al., 2005), involve type-level computation where casts are inferred by the compiler. We hope to explore similar type-level computation mechanisms in Fun in the future.

In this section, most examples written in Fun do not require programmers to write casts directly as they are automatically inferred. Two exceptions are the encodings of objects and vectors, whose elaboration rules are not implemented in the Fun yet. We demonstrate these two examples by manually adding casts. Also, to make them runnable in the prototype implementation, cast operators from the core language are allowed to be called from the surface programs.

Encoding Parametrized Algebraic Datatypes. We give an example of encoding parametrized algebraic datatypes in PITS via the $\mu$-operator and explicit casts. Note that for simplicity reasons, we use call-by-name casts to demonstrate the encoding. Call-by-value casts can similarly be used for encoding, but require a more complex treatment of recursive types, since they become values in call-by-value PITS (will be discussed later in Section 5.2). 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 a polymorphic list as

```haskell
data List a = Nil | Cons a (List a);
```

This Fun definition is translated into PITS using a Scott encoding (Mogensen, 1992) of datatypes:

```haskell
List = \mu L : * \to *. \lambda a : *. \Pi b : *. b \to (a \to L a \to b) \to b
Nil = \lambda a : *. \lambda x : a. \lambda (xs : List a).
      \text{cast}^2 [List a] (\lambda b : *. \lambda n : b. \lambda c : (a \to List a \to b). n)
Cons = \lambda a : *. \lambda x : a. \lambda (xs : List a).
      \text{cast}^2 [List a] (\lambda b : *. \lambda n : b. \lambda c : (a \to List a \to b). c \times xs)
```

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., \(\Pi\)-type. The body of \(\Pi\)-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 \text{cast}^2 is used twice here (written as \text{cast}^2): one for one-step expansion from \(\tau\) to \((\lambda a : *. \tau) a\) and the other for folding the recursive type from \((\lambda a : *. \tau) a\) to List a, where \(\tau\) is the type of \text{cast}^2 body.

We have two notable remarks from the example above. Firstly, iso-types are critical for the encoding and cannot be replaced by iso-recursive types. Since type constructors are parametrized, not only folding/unfolding recursive types, but also type-level reduction/expansion is required, which is only possible with casts. Secondly, although casts using call-by-value and call-by-name reduction are not as powerful as casts using full beta-reduction, they are still capable of encoding many useful constructs, such as algebraic datatypes and records. Nevertheless full-reduction casts enable other important applications. Some applications of full casts are discussed later (see Section 3.4).

### 3.2 Combining Algebraic Datatypes with Advanced Features

Languages like Haskell support several advanced type-level features. Such features, when used in combination with algebraic datatypes, have important applications. Next we discuss some of these features and their applications. The purpose is to show all these advanced features are encodable in PITS. For simplicity reasons, we use arrow syntax \((x : A) \to B\) as syntactic sugar for \(\Pi\)-types \(\Pi x : A. B\) in the following text.

An alternative way of encoding is to flip L and a as List = \(\lambda a : *. \mu L : *. \Pi b : *. b \to (a \to L a \to b) \to b\). However, L here is not parametric and implicitly has a fixed parameter a. Such encoding is more limited and does not work with non-uniform types where parameters can vary (e.g. PTree example in Section 3.2).
Higher-kinded Types. Higher-kinded types are type-level functions. To support higher-kinded types, languages like Haskell use core languages that account for kind expressions. The existing core language of Haskell, System FC (Sulzmann et al., 2007), is an extension of System $F_\omega$ (Girard, 1972), which natively supports higher-kinded types. We can similarly construct higher-kinded types in PITS. We show an example of encoding the functor “type-class” as a record:

```haskell
data Functor (f : * → *) = Func { fmap : (a : *) → (b : *) → (a → b) → f a → f b };
```

Note that in PITS, records are encoded using algebraic datatypes in a similar way as Haskell’s record syntax (Marlow, 2010). Here we use a record to represent a functor, whose only field is a function called $fmap$. The functor “instance” of the $Maybe$ datatype is:

```haskell
data Maybe (a : *) = Nothing | Just a;
  case x of
  Nothing ⇒ Nothing b
  | Just (z : a) ⇒ Just b (f z));
```

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

$$(f : * → *) → (a : *) → (b : *) → (a → b) → f a → f b$$

Since $Maybe$ has kind $* → *$, it is legal to apply $Func$ to $Maybe$.

Datatype Promotion. Recent versions of Haskell introduced datatype promotion (Yorgey et al., 2012), in order to allow ordinary datatypes as kinds, and data constructors as types. With the power of unified syntax, data promotion is made trivial in $Fun$. We show a representation of a labeled binary tree, where each node is labeled with its depth in the tree. Below is the definition:

```haskell
data Nat = Z | S Nat;
data PTree (n : Nat) = Empty | Fork (x : Int) (y : PTree (S n)) (z : PTree (S n));
```

Notice how the datatype $Nat$ is “promoted” to be used at the kind level in the definition of $PTree$. Next we can construct a binary tree that keeps track of its depth statically:

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

If we accidentally write the wrong depth, for example:

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

The above will fail to pass type checking.

The $PTree$ example can also be viewed as a dependent variant of nested datatypes (Bird & Meertens, 1998). For example, $Nest$ is a non-dependent nested datatype as follows (written in Haskell):
Pure Iso-Type Systems

**data Nest a = NilN | ConsN a (Nest (a,a))**

where ConsN contains a parameter with type Nest (a,a) that varies from Nest a. This is similar to Fork that contains a parameter of type PTree (S n). Notice that nested datatypes are supported in Haskell 98. Pattern matching for nested datatypes in Haskell already worked before GADTs were added to Haskell. In other words, such restricted subset of non-uniform types (i.e. nested datatypes) do not require the use of equality during pattern matching (unlike the more general case of GADTs).

**Higher-order Abstract Syntax.** Higher-order abstract syntax (Pfenning & Elliott, 1988) (HOAS) is a representation of abstract syntax where the function space of the meta-language is used to encode the binders of the object language. We show an example of encoding a simple lambda calculus:

**data Exp = Num Int | Lam (Exp → Exp) | App Exp Exp;**

Note that in the lambda constructor (Lam), the recursive occurrence of Exp appears in a negative position (i.e. in the left side of a function arrow). Systems like Coq (The Coq development team, 2016) and Agda (Norell, 2007) would reject such programs since it is well-known that such datatypes can lead to logical inconsistency. Moreover, such logical inconsistency can be exploited to write non-terminating computations, and make type checking undecidable. In contrast, Fun is able to express HOAS in a straightforward way, while preserving decidable type checking.

Using Exp we can write an evaluator for the lambda calculus. As noted by Fegaras and Sheard (1996), the evaluation function needs an extra function (reify) to invert the result of evaluation. The code for the evaluator is shown next (we omit most of the unsurprising cases; text after “--” are comments):

**data Value = VI Int | VF (Value → Value);**
**data Eval = Ev { eval : Exp → Value.reify' : Value → Exp };**
**defrec ev : Eval =**
**Ev (λ.e : Exp. case e of ... -- excerpted**
**| Lam fun ⇒ VF (λe' : Value. eval' ev (reify' ev e'))))**
**| VF fun ⇒ Lam (λe' : Exp. reify' ev (fun (eval' ev e'))));**
**def eval : Exp → Value = eval' ev;**

The definition of the evaluator is mostly straightforward. Here we create a record Eval, inside which are two name fields eval' and reify'. Similarly to the record syntax of Haskell, both eval' and reify' are also functions for projections of fields. The eval' function is conventional, dealing with each possible shape of an expression. The tricky part lies in the evaluation of a lambda abstraction, where we need a second function, called reify', of type Value → Exp that lifts values into terms. Thanks to the flexibility of the μ primitive, mutual recursion can be encoded by using records.

Evaluation of a lambda expression proceeds as follows:
Y. Yang and B. Oliveira

```python
def show = λv: Value. case v of VI n ⇒ n;
def expr = App (Lam (λf : Exp. App f (Num 42))) (Lam (λg : Exp. g));
show (eval expr) -- returns 42
```

### 3.3 Object Encodings

Casts are useful for modeling examples other than algebraic datatypes. For example, we can model a simple form of object encodings with call-by-name cast operators (Yang & Oliveira, 2017). We present an example of existential object encodings (Pierce & Turner, 1994) in Fun, which originally requires System F_Ω. First, we encode the existential type and its constructor in Fun by a standard Church encoding (Pierce, 2002) using the universal type (i.e. Π-type):

```python
def Ex = lP : ?! ?!
(z : ?) !((x : ?) ! Px ! z) ! z;
def pack = lP : ?! ?!
l1 : ?!
l2 : Pe1.
cast† [Ex P] (λz : ?. λf : (x : ?) ! Px ! z. fe1 e2);
```

where `pack` is the constructor to build an existential package. Thus, we can encode an existential type \( \exists x. A \) as `Ex (lX : ?. Pair X (X ! IX))` in Fun. The object type operator `Obj` can be encoded as follows:

```python
data Pair (A : ?) (B : ?) = MkPair {fst:A, snd:B};
def Obj = lI : ?! ?!
Ex (lX : ?. Pair X (X ! IX));
```

where `Pair A B` encodes the pair type \( A \times B \). In the definition of `Obj`, the binder `I` denotes the interface. The body is an existential type which packs a pair. The pair consists of a hidden state (with type \( X \)) and methods which are functions depending on the state (with type \( X \rightarrow I X \)). For a concrete example of objects, we use the interface of cell objects (Bruce et al., 1999):

```python
data Cell (X : *) = MkCell {get:Int, set: Int → X, bump:X};
```

The interface indicates that a cell object consists of three methods: a getter `get` to return the current state, a setter `set` to return a new cell with a given state, and `bump` to return a new cell with the state increased by one.

We can define a cell object `c` as follows:

```python
data Var = MkVar {getVar: Int};
def CellT = λX : *. Pair X (X ! Cell X);
def pair = MkPair Var (Var ! Cell Var)
(MkVar 0) -- Initial state
(λs : Var. MkCell Var -- Methods
(getVar s) -- Method get
(λn : Int.MkVar n) -- Method set
(MkVar (getVar s + 1))); -- Method bump
def p = pack CellT Var (cast† [CellT Var] pair);
def c = cast† [Obj Cell] p;
```
The body of object \( c \) is an existential package \( p \) of type \( \exists X.\text{Pair} \ X \ (X \rightarrow \text{Cell} \ X) \) built by the \emph{pack} operator. The first parameter of \emph{pack} is \( \text{CellT} \) that represents the body of the existential type. The second parameter is the integer variable type \( \text{Var} \) which corresponds to the existential binder \( X \). The third parameter has type \( \text{CellT} \ \text{Var} \) which can be reduced to a pair type \( \text{Var} \times (\text{Var} \rightarrow \text{Cell} \ \text{Var}) \) which is defined in the definition \emph{pair} constructed by \emph{MkPair}. The first component of the \emph{pair} is the initial hidden state \( \text{MkVar} \ 0 \). The second component is a function containing three methods that are defined in a record by \emph{MkCell} and abstracted by the state variable \( s \). The definition of the three methods follows the cell object interface \emph{Cell}.

Note that we have two \emph{cast} operators here: one over the existential package \( p \) and another over the \emph{pair}. Due to the lack of a conversion rule in PITS, the desired type of the object \( c \) (i.e. \( \text{Obj Cell} \)) is an application, which is different from the type of the existential package (i.e. \( \text{Ex CellT} \)). Noticing that

\[
\text{Obj Cell} \leftarrow \text{Ex} \ (\lambda X \cdot \star, \text{Pair} \ X \ (X \rightarrow \text{Cell} \ X)) = \text{Ex CellT}
\]

We can use \emph{cast} to do one-step type expansion for the package. Similarly, the second \emph{cast} operator used in the third parameter of \emph{pack} converts the pair type into \( \text{CellT} \ \text{Var} \). We emphasize that the object encoding example exploits fundamental features of PITS, namely higher-kinded types, higher-order polymorphism and explicit casts. The absence of a conversion rule does not prevent the object encoding because the required type-level computation is recovered by explicit casts.

### 3.4 Fun with Full Reduction

So far all the examples can be encoded in \emph{Fun} with casts using weak-head reduction. However for some applications full reduction is needed at the type level. In this subsection particularly, we show one such application. For brevity, we move types of arguments in \emph{def} bindings to top-level annotations without repeating them in \( \lambda \)-binders, e.g., we change

\[
\text{def} \ f : \lambda x : \text{Int} \ 1 \text{ into}
\]

\[
\text{def} \ f : \text{Int} \rightarrow \text{Int} = \lambda x . \ 1
\]

\textbf{Leibniz Equality, Kind Polymorphism and Vectors.} One interesting type-level feature of GHC Haskell is generalized algebraic datatypes, or GADTs (Xi \emph{et al.}, 2003; Cheney \& Hinze, 2003; Peyton Jones \emph{et al.}, 2004). GADTs require a non-trivial form of explicit type equality, which is built in Haskell’s core language (System FC (Sulzmann \emph{et al.}, 2007)), called a \emph{coercion}. PITS does not have such built-in equality. However a form of equality can be encoded using \textbf{Leibniz Equality} (Cheney \& Hinze, 2002):

\[
\texttt{data Eq (k:*) (a:k) (b:k) =}
\texttt{Proof \{ subst : (f:k \rightarrow *) \rightarrow f \ a \rightarrow f \ b \};}
\]

Note that the definition uses \textit{kind polymorphism}: the kind of types \( a \) and \( b \) is \( k \), which is polymorphic and not limited to \( * \). For brevity, we use \( a \equiv b \) to denote \( \text{Eq} \ k \ a \ b \) by omitting the kind \( k \). Then we can encode a GADT, for example, length-indexed list (or \emph{vector}) as follows:
However, it is difficult to use such encoding approach to express the injectivity of constructors (Cheney & Hinze, 2003), e.g., deducing \( n \equiv m \) from \( Sn \equiv Sm \). It would be challenging to encode the tail function of a vector:

```python
def tail: (a:*) \to (n:Nat) \to (v:Vec a (Sn)) \to Vec a n =
  \lambda a. \lambda n. \lambda v. case v of Cons m p x xs => xs; -- ill-typed
```

The case expression above is ill-typed: \( xs \) has the type \( Vec a m \), but the function requires the case branch to return \( Vec a n \). To convert \( xs \) to the correct type, we need to show \( n \equiv m \).

The auxiliary function "\( \text{lift}^2 \)" lifts the type of equality proof \( p \) from \( Sn \equiv Sm \) to \( \text{predNat} (Sn) \equiv \text{predNat} (Sm) \). Then \( \text{cast}_0 \) converts it to \( Eq \text{Nat} n m \) (type annotations are omitted for brevity):

```python
p : Sn \equiv Sm
lift predNat p : predNat (Sn) \equiv predNat (Sm)
cast_0 (lift predNat p) : n \equiv m
```

Now we can write a well-typed version of \( \text{tail} \):

```python
def castVec: (a:*) \to (n:Nat) \to (m:Nat) \to
  Eq Nat n m \to Vec a m \to Vec a n =
  \lambda a. \lambda n. \lambda m. \lambda p. to (Vec a n) (Vec a m) (lift Nat \ast n m (Vec a) p);
def tail: (a:*) \to (n:Nat) \to (v:Vec a (Sn)) \to Vec a n =
  \lambda a. \lambda n. \lambda v. case v of Cons m p x xs => castVec a n m (injNat n m p) xs;
```

\(^2\) The definition of \( \text{lift} \) is omitted and available from online materials.
We use \texttt{injNat} to obtain a proof \( n \equiv m \) from \( S \ n \equiv S \ m \). Finally, we call an auxiliary function \texttt{castVec} that uses the proof \( n \equiv m \) to convert \( xs \) from \( \text{Vec} \ a \ m \) to \( \text{Vec} \ a \ n \). The definition of \texttt{castVec} relies on another auxiliary function "to"\(^3\) that unfolds the Leibniz equality \( \text{Vec} \ a \ n \equiv \text{Vec} \ a \ m \), lifted from \( n \equiv m \), to a function with type \( \text{Vec} \ a \ m \rightarrow \text{Vec} \ a \ n \).

Note that \texttt{Fun} is not logically consistent and does not check whether the proof is terminating. Nonetheless, we can manually ensure that the injectivity proof \texttt{injNat} used in \texttt{tail} is valid. There are no recursive terms in \texttt{injNat}, which could be bogus proofs. Though \texttt{predNat} is a partial function, it is always applied to numbers with the form \( S \ n \) and never goes to the invalid branch. Finally, while this encoding deals with injectivity, an encoding of equality still has a computational cost. Therefore, a motivation to natively support equality constraints instead (as done for example in Haskell (Sulzmann et al., 2007)) is to have equality without any computational costs, since equality constraints can simply be erased.

4 Call-by-name Pure Iso-Type Systems

We formally present the first variant of Pure Iso-Type Systems. PITS is very close to Pure Type Systems (PTS) (Barendregt, 1992), except for two key differences: the existence of cast operators and general recursion. In this section, we focus on the call-by-name variant of PITS, which uses a call-by-name weak-head reduction strategy in casts. We show type safety for any PITS and decidability of type checking for a particular subset, i.e., functional PITS (see Definition 4.1). One important remark is that the dynamic semantics of call-by-name PITS is given by a direct small-step operational semantics, and type-safety is proved using the usual preservation and progress theorems. Full proofs of the metatheory are mechanized in Coq and can be found in the online repository.

4.1 Syntax

Fig. 1 shows the syntax of PITS, including expressions, values and contexts. Like Pure Type Systems (PTS), PITS uses a unified representation for different syntactic levels. There is no syntactic distinction between terms, types or kinds/sorts. Such unified syntax brings economy for type checking, since one set of typing rules can cover all syntactic levels. As in PTS, PITS contains a set of constants called \texttt{Sorts}, e.g., *, \( \square \), denoted by metavariable \( s \).

By convention, we use metavariables \( A, B \), etc. for an expression on the type-level position and \( e \) for one on the term level. We use \( A \rightarrow B \) as a syntactic sugar for \( \Pi x : A. \ B \) if \( x \) does not occur free in \( B \).

\textbf{Cast Operators.} We introduce two new primitives \texttt{cast\(_\uparrow\)} and \texttt{cast\(_\downarrow\)} (pronounced as “cast up” and “cast down”) to replace the implicit conversion rule of PTS with one-step explicit type conversions. The cast operators perform two directions of conversion: \texttt{cast\(_\uparrow\)} is for the one-step reduction of types, and \texttt{cast\(_\downarrow\)} is for the one-step expansion. The \texttt{cast\(_\downarrow\)} construct needs a type annotation \( A \) as the result type of one-step expansion for disambiguation.

\(^3\) The definition of \texttt{to} is omitted and available from online materials.
Expressions $e, A, B ::= x | s | e_1 e_2 | \lambda x : A. e | \Pi x : A. B$

| Values $v ::= x | \lambda x : A. e | \Pi x : A. B | \text{cast}_1 [A] e$
| Contexts $\Gamma ::= \emptyset | \Gamma, x : A$

Syntactic Sugar

$A \rightarrow B \triangleq \Pi x : A. B$ where $x \not\in \text{FV} (B)$

$\text{cast}^n [A_1] e \triangleq \text{cast}_1 [A_1](\text{cast}_1 [A_2](\ldots (\text{cast}_1 [A_n] e)\ldots))$

where $A_1 \hookrightarrow A_2 \hookrightarrow \ldots \hookrightarrow A_n$

$\text{cast}^n e \triangleq \underbrace{\text{cast}_1(\ldots (\text{cast}_1 e)\ldots)}_{n}$

Fig. 1. Syntax of call-by-name PITS

while $\text{cast}_1$ does not, since the result type of one-step reduction can be uniquely determined as discussed in Section 4.5.

We use syntactic sugar $\text{cast}^n$ and $\text{cast}^n_#$ to denote $n$ consecutive cast operators (see Fig. 1). Alternatively, one can introduce them as built-in operators and treat one-step casts as syntactic sugar instead. Though using built-in $n$-step casts can reduce the number of individual cast constructs, we do not adopt such alternative design in order to simplify the discussion of metatheory. Note that $\text{cast}^n$ is simplified to take just one type parameter, i.e., the last type $A_1$ of the $n$ cast operations. Due to the determinacy of one-step reduction (see Lemma 4.2), the intermediate types can be uniquely determined and left out.

General Recursion. We use the $\mu$-operator to uniformly represent recursive terms and types. The expression $\mu x : A. e$ can be used on the type level as a recursive type, or on term level as a fixpoint that is possibly non-terminating. For example, $A$ can be a single sort $s$, as well as a function type such as $\text{Int} \rightarrow \text{Int}$ or $s_1 \rightarrow s_2$.

4.2 Operational Semantics

Fig. 2 shows the small-step, call-by-name operational semantics. Three base cases include rule R-BETA for beta reduction, rule R-MU for recursion unrolling and rule R-CASTELIM for cast canceling. Two inductive cases, rule R-APP and rule R-CASTDN, define reduction at the head position of an application, and the inner expression of cast terms, respectively. Note that rule R-CASTELIM and rule R-CASTDN do not overlap because in the former rule, the inner term of $\text{cast}_1$ is a value (see Fig. 1), i.e., $\text{cast}_1 [A] e$. In rule R-CASTDN, the inner term is reducible and cannot be a value.

The reduction rules are called weak-head since only the head term of an application can be reduced, as indicated by the rule R-APP. Reduction is also not allowed inside the $\lambda$-term and $\Pi$-term which are both defined as values. To remove such restrictions, we need the parallel reduction, as will be discussed in Section 6. Weak-head reduction rules are used for both type conversion and term evaluation. 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.
Pure Iso-Type Systems

Fig. 2. Operational semantics of call-by-name PITS

Fig. 3. Typing rules of call-by-name PITS

4.3 Typing

Fig. 3 gives the syntax-directed typing rules of PITS, including rules of context well-formedness \( \vdash \Gamma \) and expression typing \( \Gamma \vdash e : A \). Note that there is only a single set of rules for expression typing, as there is no distinction of different syntactic levels. Most typing rules are quite standard. We write \( \vdash \Gamma \) if a context \( \Gamma \) is well-formed. We use \( \Gamma \vdash A : s \) to check if \( A \) is a well-formed type.

PITS is a family of type systems similarly to PTS, parametrized by the axiom set \( \mathcal{A} \subset (\text{Sorts} \times \text{Sorts}) \) for typing sorts and rule set \( \mathcal{R} \subset (\text{Sorts} \times \text{Sorts} \times \text{Sorts}) \) for checking well-formedness of \( \Pi \)-types. Rule T-AX checks whether sort \( s_1 \) can be typed by sort \( s_2 \) if \( (s_1,s_2) \in \mathcal{A} \) holds. Rule T-VAR checks the type of variable \( x \) from the valid context. Rule T-APP and rule T-ABS check the validity of application and abstraction respectively. Rule T-PROD checks the type well-formedness of the dependent function type by checking
Y. Yang and B. Oliveira

if \((s_1, s_2, s_3) \in R\). Rule T-MU checks the validity of a recursive term. It ensures that the recursion \(\mu x : A. e\) should have the same type \(A\) as the binder \(x\) and also the inner term \(e\).

The Cast Rules. We focus on the rule T-CASTUP and rule T-CASTDN that define the semantics of cast operators and replace the conversion rule of PTS. The relation between the original and converted type is defined by one-step call-by-name reduction (see Fig. 2). For example, given a judgment \(\Gamma \vdash e : A_2\) and relation \(A_1 \leftrightarrow A_2 \leftrightarrow A_3\), cast\(^+\) \([A_1]\) \(e\) expands the type of \(e\) from \(A_2\) to \(A_1\), while cast\(^-\) \(e\) reduces the type of \(e\) from \(A_2\) to \(A_3\). We can formally give the typing derivations of the examples in Section 2.3:

\[
\begin{align*}
\Gamma \vdash e : (\lambda x : \ast. x) Int & \quad \Gamma \vdash 3 : Int \\
(\lambda x : \ast. x) Int \leftrightarrow Int & \quad (\lambda x : \ast. x) Int \leftrightarrow Int \\
\Gamma \vdash (\text{cast}^+ e) : Int & \quad \Gamma \vdash (\text{cast}^- [ (\lambda x : \ast. x) Int ] 3) : (\lambda x : \ast. x) Int
\end{align*}
\]

Importantly, in PITS 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.

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

4.4 The Two Faces of Recursion

One key difference from PTS is that PITS supports general recursion for both terms and types. We discuss general recursion on two levels and show how iso-types generalize iso-recursive types.

Term-level Recursion. In PITS, the \(\mu\)-operator works as a fixpoint on the term level. By rule R-MU, evaluating a term \(\mu x : A. e\) will substitute all \(x\)'s in \(e\) with the whole \(\mu\)-term itself, resulting in the unrolling \(e[x \mapsto \mu x : A. e]\). The \(\mu\)-term is equivalent to a recursive function that should be allowed to unroll without restriction.

Recall the factorial function example in Section 2.4. By rule T-MU, the type of \(\text{fact}\) is \(\text{Int} \rightarrow \text{Int}\). Thus we can apply \(\text{fact}\) to an integer. Note that by rule R-MU, \(\text{fact}\) will be unrolled to a \(\lambda\)-term. Assuming the evaluation of if-then-else construct and arithmetic expressions follows the one-step reduction, we can evaluate the term \(\text{fact} 3\) as follows:

\[
\begin{align*}
\text{fact} 3 & \\
\quad \leftrightarrow (\lambda x : \text{Int}. \text{if } x == 0 \text{ then } 1 \text{ else } x \times \text{fact} (x - 1)) 3 & \quad \text{-- by rule R-App} \\
\quad \leftrightarrow \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact} (3 - 1) & \quad \text{-- by rule R-BETA} \\
\quad \leftrightarrow \ldots \leftrightarrow 6
\end{align*}
\]

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

**Type-level Recursion.** On the type level, $\mu x : A. e$ works as an iso-recursive type (Crary et al., 1999), 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 that there exist expressions $e_1$ and $e_2$ such that $e_1 : \mu x : A. B$ and $e_2 : B[x \mapsto \mu x : A. B]$, we have the following typing results:

$$\begin{align*}
\text{unfold } e_1 & : B[x \mapsto \mu x : A. B] \\
\text{fold } [\mu x : A. B] e_2 & : \mu x : A. B
\end{align*}$$

by applying standard typing rules of iso-recursive types (Pierce, 2002):

$$\begin{align*}
\Gamma \vdash e_1 : \mu x : A. B & \\
\Gamma \vdash \mu x : A. B : s & \\
\Gamma \vdash e_2 : B[x \mapsto \mu x : A. B] & \\
\Gamma \vdash \text{fold } [\mu x : A. B] e_2 : \mu x : A. B
\end{align*}$$

We have the following relation between types of $e_1$ and $e_2$ witnessed by fold and unfold:

$$\mu x : A. B \xrightarrow{\text{unfold}} B[x \mapsto \mu x : A. B] \xleftarrow{\text{fold}} [\mu x : A. B]$$

In PITS, we do not need to introduce fold and unfold operators, because with the rule R-MU, cast$_1$ and cast$_2$ generalize fold and unfold, respectively. 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 R-MU:

$$\mu x : A. B \leftrightarrow B[x \mapsto \mu x : A. B]$$

By applying rule T-CastUP and rule T-CastDN, we can obtain the following typing results:

$$\begin{align*}
\text{cast}_1 e_1 & : B[x \mapsto \mu x : A. B] \\
\text{cast}_2 [\mu x : A. B] e_2 & : \mu x : A. B
\end{align*}$$

Thus, cast$_1$ and cast$_2$ witness the isomorphism between the original recursive type and its unrolling, behaving in the same way as fold and unfold in iso-recursive types.

$$\mu x : A. B \xrightarrow{\text{cast}_1} B[x \mapsto \mu x : A. B] \xleftarrow{\text{cast}_2} [\mu x : A. B]$$

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 3.1).

**4.5 Metatheory of Call-by-name PITS**

We now discuss the metatheory of call-by-name PITS. We focus on two properties: the decidability of type checking and the type safety of the language. Firstly, we show that
type checking for a functional subset (Siles & Herbelin, 2012) of PITS is decidable without requiring strong normalization. Secondly, any PITS is type-safe, proven by subject reduction and progress lemmas (Wright & Felleisen, 1994).

**Decidability of Type Checking for Functional PITS.** We limit the discussion in this paragraph to a subclass of PITS, functional PITS:

**Definition 4.1**
A PITS is functional if:

1. for all $s_1, s_2, s_2', s_0, s_0'$, if $(s_1 \rightarrow s_2) \in \mathcal{A}$ and $(s_1 \rightarrow s_0) \in \mathcal{A}$, then $s_2 \equiv s_0'$.
2. for all $s_1, s_2, s_3, s_3', s_0, s_0'$, if $(s_1 \rightarrow s_2 \rightarrow s_3) \in \mathcal{R}$ and $(s_1 \rightarrow s_2 \rightarrow s_0) \in \mathcal{R}$, then $s_3 \equiv s_0'$.

Such definition is similar to the one of functional PTS (Siles & Herbelin, 2012), or singly-sorted PTS (Barendregt, 1992). Functional PITS enjoys uniqueness of typing, i.e., typing result is unique up to alpha equality:

**Lemma 4.1** (Uniqueness of Typing for Functional PITS)
In any functional PITS, if $\Gamma \vdash e : A$ and $\Gamma \vdash e : B$, then $A \equiv B$.

For simplicity reasons, we only discuss decidability for functional PITS, where the proof can be significantly simplified by the uniqueness of typing lemma. For non-functional PITS, one may follow the proof strategy similarly used in non-functional PTS (van Benthem Jutting, 1993), by proving the “Uniqueness of Domains” lemma instead. We leave the decidability proof for non-functional PITS as future work.

For functional PITS, 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 rule T-CASTU and rule T-CASTDN. Both rules contain a premise that needs to judge if two types $A$ and $B$ follow the one-step reduction, i.e., if $A \rightarrow B$ holds. We show that $B$ is unique with respect to the one-step reduction, or equivalently, reducing $A$ by one step will get only a sole result $B$. Such property is given by the following lemma:

**Lemma 4.2** (Determinacy of One-step Call-by-name Reduction)
If $e \rightarrow e_1$ and $e \rightarrow e_2$, 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 $\mu x : A. e$, by rule R-MU, there always exists a unique term $e' \equiv e[x \mapsto \mu x : A. e]$ such that $\mu x : A. e \rightarrow e'$. With this result, we show that it is decidable to check whether the one-step relation $A \rightarrow B$ holds. We first reduce $A$ by one step to obtain $A'$ (which is unique by Lemma 4.2), and compare if $A'$ and $B$ are syntactically equal. Thus, we can further show type checking cast-terms is decidable.

By the definition of functional PITS, checking the type of sorts and well-formedness of $\Pi$-types are decidable. For other cases, type checking is decidable by the induction hypothesis and uniqueness of typing (see Lemma 4.1). Thus, we can conclude the decidability of type checking:

**Theorem 4.1** (Decidability of Type Checking for Functional PITS)
In any functional PITS, given a well-formed context $G$ and a term $e$, it is decidable to determine if there exists $A$ such that $G \vdash e : A$.

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 call-by-name reduction, which already has a decidable checking algorithm according to Lemma 4.2. We do not need to further require the normalization of terms. This is different from the proof for PTS which requires the language to be strongly normalizing (van Benthem Jutting, 1993). In PTS the conversion rule needs to examine the beta equivalence of terms, which is decidable only if every term has a normal form.

Type Safety for all PITS. Type safety holds for any PITS, not just functional PITS. The proof of the type safety is by showing subject reduction and progress lemmas (Wright & Felleisen, 1994):

**Theorem 4.2 (Subject Reduction of Call-by-name PITS)**
If $G \vdash e : A$ and $e \Rightarrow e'$ then $G \vdash e' : A$.

**Theorem 4.3 (Progress of Call-by-name PITS)**
If $\varnothing \vdash e : A$ then either $e$ is a value $v$ or there exists $e'$ such that $e \Rightarrow e'$.

The proof of subject reduction is straightforward by induction on the derivation of $G \vdash e : A$ and inversion of $e \Rightarrow e'$. Some cases need supporting lemmas: rule R-C

**AST**

**ELIM**

**AST**

**E**

**LIM**

requires Lemma 4.2; rule R-B

**ETA**

and rule R-M

**U**

require the following substitution lemma:

**Lemma 4.3 (Substitution of Call-by-name PITS)**
If $G_1, x : B, G_2 \vdash e_1 : A$ and $G_1 \vdash e_2 : B$, then $G_1, G_2[x \mapsto e_2] \vdash e_1[x \mapsto e_2] : A[x \mapsto e_2]$.

The proof of progress is also standard by induction on $\varnothing \vdash e : A$. Notice that cast$_T$ [$A$] $e$ is a value, while cast$_T$ $e_1$ is not: by rule R-CASTDN, $e_1$ will be constantly reduced until it becomes a value that could only be in the form cast$_T$ [$A$] $e$ by typing rule T-CASTDN. Then rule R-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 $\lambda$-term but not a cast$_T$-term. Otherwise, suppose $e_1$ has the form cast$_T$ [$\Pi x : A. B$] $e'_1$. By inversion, we have $\varnothing \vdash e'_1 : A'$ and $\Pi x : A. B \Rightarrow e'_1$. But such $A'$ does not exist because $\Pi x : A. B$ is a value which is not reducible.

5 Call-by-value Pure Iso-Type Systems

Pure Iso-Type Systems enjoy the flexibility of choosing different reduction rules for type conversion or term evaluation. In this section, we present another variant of PITS which uses call-by-value reduction, a more commonly used reduction strategy. All metatheory presented in Section 4, including type safety and decidability of typing, still holds for this variant. Call-by-value is interesting because, for applications, the arguments are reduced before beta reduction. Such reduction is problematic for dependent functions types in a setting with iso-types, since it may endanger type safety. We address this problem by using a form of value restriction, inspired by previous work (Swamy et al., 2011; Sjöberg et al., 2012). Full proofs are mechanized in Coq and can be found in the online repository.
5.1 Value Restriction

Call-by-value reduction (\(\rightarrow_v\)) requires the argument of beta reduction to be a value (\(v\)). A typical leftmost reduction strategy of an application is that, one first reduces the function part to a value, then further reduces the argument. Such process is witnessed by the following reduction rules:

\[
\text{RV-BETA} \\
(\lambda x : A. e) \, v \rightarrow_v e[x \mapsto v] \\
\text{RV-APPL} \\
e_1 \rightarrow_v e'_1 \\
e_2 \rightarrow_v e'_2 \\
\rightarrow_v e e' = v e \rightarrow_v v e'
\]

A function in PITS can be dependent. That is the function type can depend on the argument. For example, suppose that \(f\) is a dependent function generating a length-indexed list with given length \(n\), with the following type:

\[f : (\Pi n : \text{Int}. \text{Vec} \, n)\]

By rule RV-APPR, the reduction \(f(1 + 1) \rightarrow_v f \, 2\) holds. However, the types of two sides of the reduction are different:

\[f(1 + 1) : \text{Vec} \, (1 + 1)\]
\[f \, 2 : \text{Vec} \, 2\]

Notice that PITS does not contain an implicit conversion rule. Without an explicit type conversion, subject reduction does not hold. There are at least two possible ways to deal with this issue: 1) introducing a type cast for reduction; 2) requiring the dependent function to be applied to a value only. For simplicity reasons, we choose the second method: a value restriction. The first method entangles types with reduction, which makes the semantics more complicated. The value restriction is also used by several existing call-by-value calculi to ensure type safety or simplify the design, e.g., by Zombie (Sjöberg et al., 2012; Casinghino et al., 2014) to prevent bogus proofs that are non-terminating, and by DOT (Amin et al., 2012; Amin et al., 2016), a core language of Scala (Odersky et al., 2004), to simplify the formalization of path-dependent types (Amin et al., 2014).

Typing Rules with Value Restriction. Now we have two typing rules for the function application:

\[
\text{TV-APPV} \\
\Gamma \vdash v : \Pi x : A. B \\
\Gamma \vdash e : A \\
\Gamma \vdash v \, e : B[x \mapsto v]
\]
\[
\text{TV-APP} \\
\Gamma \vdash e_1 : A \rightarrow B \\
\Gamma \vdash e_2 : A \\
\Gamma \vdash e_1 \, e_2 : B
\]

If a function is dependent the argument must be a value (\(v\)). Otherwise, the function is non-dependent and there is no restriction on its argument. Recall that the arrow type is syntactic sugar for the non-dependent Pi-type (see Fig. 1). Note that these two typing rules overlap: the Pi-type of \(e\) in rule TV-APPV could be non-dependent. In such case, \(B = B[x \mapsto v]\) and rule TV-APPV has the same typing result as rule TV-APP. Thus, such overlapping does not cause any determinacy issues.
Two Sides of Value Restriction. Imposing such value restriction in PITS has both pros and cons. On one side, type-safety proofs are quite simple (see Section 5.3). We can safely rule out the case that breaks type preservation when reducing the argument of a dependent function application. Recalling the example above, a reduction like \( f (1 + 1) \mapsto v f 2 \) is not possible since \( f (1 + 1) \) will be rejected by the type system in the first place. The argument \( (1 + 1) \) of the dependent function \( f \) is not a value, so \( f (1 + 1) \) is not a well-typed term. Thus, if a function is applied to a reducible argument it must be a non-dependent function in order to ensure type preservation.

Though users can easily write \( f 2 \) as a workaround to satisfy the type system instead of \( f (1 + 1) \), there is no alternative way to express terms such as \( f (x + y) \) where the argument cannot be reduced to a value. Thus, the value restriction makes the type system more restrictive on dependent function applications — users can only provide values but not arbitrary arguments to dependent functions. Nonetheless, there is no restriction on applying non-dependent functions to arguments, e.g., \( \text{id} (x + y) \) where \( \text{id} = \lambda z : \text{Int}. \ z \).

In other words, there is no loss of expressiveness with respect to non-dependently typed programming. One nice aspect of call-by-value casts is that they can be erased after type-checking (i.e. computational irrelevance, see Section 7.1).

Also, the value restriction does not make call-by-value PITS necessarily less expressive than call-by-name PITS. Though dependent applications such as \( f (1 + 1) \) can be written in call-by-name PITS, the argument \( (1 + 1) \) may never be reduced and is always kept as is. Thus, one may not be able to prove equality between \( f (1 + 1) \) and \( f 2 \) in call-by-name PITS. On the other hand, one can reduce \( \text{id} (1 + 1) \) to \( \text{id} 2 \) in call-by-value PITS but not in call-by-name PITS.

Alternative to the Value Restriction. Instead of the value restriction, one could simultaneously add casts when reducing dependent function applications. Supposing we drop the value restriction, for the same example of \( f \), noticing that \( \text{Vec} (1 + 1) \mapsto v \text{Vec} 2 \), we can obtain

\[
f 2 \quad : \text{Vec} 2 \\
\text{cast}_v [\text{Vec} (1 + 1)] (f 2) \quad : \text{Vec} (1 + 1)
\]

Then, \( f (1 + 1) \mapsto v \text{cast}_v [\text{Vec} (1 + 1)] (f 2) \) preserves the type with an extra \( \text{cast}_v \). However, such reduction relation involves types due to the annotation of casts. Moreover, call-by-value casts may not be expressive enough for certain type conversions, such as conversions inside binders. Parallel reduction and full casts may be required, as will be discussed in Section 6. For simplicity reasons, we leave this extension as future work and stick to value restriction for a simpler meta-theory.

5.2 Reduction with Open Terms

In call-by-value PITS, cast operators also use one-step call-by-value reduction to perform type conversions. Open terms that contain free variables may occur during reduction, e.g., \( (\lambda x : \text{Int}. \ x) y \), where \( y \) is a free variable. Using the rule \( \text{RV-BETA} \), the reduction can only be performed if \( y \) is a value. To allow beta reduction of such open terms, we allow variables as values. Traditional call-by-value calculi do not have such definitions, since reduction
is used for term evaluation but not type conversion. Nevertheless, several call-by-value calculi that involve type conversion allow treating variables as values, such as several open call-by-value calculi (Accattoli & Guerrieri, 2016) (including Fireball Calculus (Paolini & Della Rocca, 1999)) and Zombie (Sjöberg et al., 2012; Casinghino et al., 2014). To make such definition work, we need to ensure that a variable is substituted with a value. The rule RV-BETA already ensures such requirement.

Recursion and Recursive Types. For a recursive term $\mu x : A. e$, its unrolling has the form $e[x \mapsto \mu x : A. e]$ such that the substituted term is the term itself. Thus, we need to treat $\mu$-terms as values as in traditional call-by-value settings. One consequence is that a $\mu$-term now is only unrolled when it is placed at the function part of an application, or inside cast $\#/

RV-MU RV-CAST-D-MU

| $\mu x : A. e \mapsto v (e[x \mapsto \mu x : A. e])$ | $\mu x : A. e \mapsto v \text{ cast}_\# (e[x \mapsto \mu x : A. e])$ |

Thus, $\mu$-terms only represents term-level recursive functions, as in most call-by-value languages. They cannot directly represent recursive types at the same time. This is different from the call-by-name PITS where $\mu$-terms are both term-level fixpoints and recursive types, since substituted terms are not necessarily values. Nevertheless, one can still recover recursive types in call-by-value PITS by feeding the type-level recursive function a dummy argument, e.g., the unit value $()$:

$$PITS \text{ Variant} \quad \text{Recursive Type} \quad \text{Reduction}$$

Call-by-name $f = \mu y : A. e$

Call-by-value $f' = \mu x : \text{Unit} \rightarrow A. e[y \mapsto x()]$

where $x$ does not occur free in $e$. The recursive type $f$ in call-by-name PITS can be simulated by a type-level recursive function $f'$ applied to a dummy argument, i.e., $f'(())$.

Finally, we show the full definition of call-by-value PITS in Fig. 4. The changes from the call-by-name variant are highlighted.

5.3 Metatheory

All the metatheory of call-by-name PITS still holds in the call-by-value variant, including two key properties: type safety and decidability of type checking. The proofs are almost the same. The only relevant change is the statement of the substitution lemma:

**Lemma 5.1 (Substitution of Call-by-value PITS)**
If $\Gamma_1, x : B, \Gamma_2 \vdash e : A$ and $\Gamma_1 \vdash v : B$, then $\Gamma_1, \Gamma_2[x \mapsto v] \vdash e[x \mapsto v] : A[x \mapsto v]$.

We now require substituted terms to be values. With such a change, type preservation of reducing open terms is possible, as discussed in Section 5.2. Such restricted substitution lemma is sufficient for proving subject reduction, because all substituted terms are values in reduction rules (see Fig. 4). The subject reduction and progress lemmas can be proved in a similar way to call-by-name PITS:

**Theorem 5.1 (Subject Reduction of Call-by-value PITS)**
If $\Gamma \vdash e : A$ and $e \mapsto e'$ then $\Gamma \vdash e' : A$. 
Lemma 5.2

If $\emptyset \vdash_{v} e : A$ then either $e$ is a value $v$ or there exists $e'$ such that $e \rightsquigarrow_{v} e'$.

Like call-by-name reduction, the one-step call-by-value reduction is deterministic:

Theorem 5.2 (Progress of Call-by-value PITS)

If $\emptyset \vdash_{v} e : A$ then either $e$ is a value $v$ or there exists $e'$ such that $e \rightsquigarrow_{v} e'$.

Like call-by-name reduction, the one-step call-by-value reduction is deterministic:

Lemma 5.2 (Determinacy of One-step Call-by-value Reduction)
Y. Yang and B. Oliveira

If $e \mapsto_{\nu} e_1$ and $e \mapsto_{\nu} e_2$, then $e_1 \equiv e_2$.

Similarly, for functional PITS, we have typing uniqueness and decidable type checking:

**Lemma 5.3 (Uniqueness of Typing for Functional PITS)**
In any functional PITS, if $\Gamma \vdash_{\nu} e : A$ and $\Gamma \vdash_{\nu} e : B$, then $A \equiv B$.

**Theorem 5.3 (Decidability of Type Checking for Functional PITS)**
In any functional PITS, given a well-formed context $\Gamma$ and a term $e$, it is decidable to determine if there exists $A$ such that $\Gamma \vdash_{\nu} e : A$.

### 6 Iso-Types with Full Casts

In Sections 4 and 5, we have introduced two variants of PITS that use one-step call-by-name/value reduction for both term evaluation and type conversion. The use of those reduction strategies simplifies the design and metatheory, at the cost of some expressiveness. To gain extra expressiveness, we take one step further to generalize casts with full reduction. In this section, we present a third variant of PITS called full PITS, where casts use a decidable parallel reduction relation for type conversion. The trade-off is some extra complexity in the metatheory. We show that full PITS has decidable type checking and type safety that holds up to erasure of casts. The proofs and metatheory design is inspired by approaches used in Zombie (Sjöberg & Weirich, 2015) and Guru (Stump et al., 2008).

#### 6.1 Full Casts with Parallel Reduction

Cast operators in call-by-name/value PITS use the same one-step reduction as term evaluation for type-level computation. We refer to them as weak casts, because they lack the ability to do full type-level computation where reduction can occur at any position of terms. For example, weak casts cannot convert the type $\text{Vec} (1+1)$ to $\text{Vec} 2$, because 1) for call-by-name reduction, the desired reduction is at the non-head position; 2) for call-by-value reduction, the term is rejected due to the value restriction. Thus, we generalize weak casts to full casts ($\text{cast}_g$ and $\text{cast}_p$) utilizing a one-step decidable parallel reduction ($\mapsto_p$) relation for type conversion. Fig. 5 shows the definition of $\mapsto_p$. This relation allows reducing terms at any position, including non-head positions or inside binders, e.g., $\lambda x : \text{Int}. \ 1+1 \mapsto_p \lambda x : \text{Int}. 2$. Thus full type-level computation for casts is enabled.

There are three remarks for parallel reduction worth mentioning. Firstly, parallel reduction is defined up to erasure denoted by $|e|$ (see Fig. 6), a process that removes all casts from term $e$. We also extend erasure to context denoted by $|\Gamma|$. It is feasible to define parallel reduction only for erased terms because casts in full PITS (also call-by-value PITS) are only used to ensure the decidability of type checking and have no effect on dynamic semantics, thus are computationally irrelevant (will be discussed in Section 7.1). We use metavariables $r$ and $R$ to range over erased terms and types, respectively. The only syntactic change of erased terms is that there is no cast. The type system after erasure is essentially a variant of PTS with recursion. We call it $\text{PTS}_\mu$. The syntax and semantics of $\text{PTS}_\mu$ is shown in Fig. 7.
Pure Iso-Type Systems

(Decidable Parallel Reduction)

\[
\begin{align*}
\text{P-RED} & \quad (\lambda x : R \cdot r_1) \triangleright_p r_2 \rightarrow_p r_1[r_1 \rightarrow r_2] \quad \text{P-VAR} & \quad \lambda x : R \cdot x \rightarrow_p x \quad \text{P-SORT} & \quad s \rightarrow_p s \quad \text{P-APPR} & \quad r_1 \triangleright_p r'_1 \quad r_2 \triangleright_p r'_2 \rightarrow_p r'_1 r'_2 \\
\text{P-ABS} & \quad R \rightarrow_p R' \quad r \rightarrow_p r' \rightarrow_p R' \quad \Pi x : R_1, R_2 \rightarrow_p \Pi x : R'_1, R'_2 \\
\text{P-PROD} & \quad R_1 \rightarrow_p R'_1 \quad R_2 \rightarrow_p R'_2 \rightarrow_p R' \quad \Pi x : A, B \rightarrow_p \Pi x : A, B' \\
\text{P-MURED} & \quad \mu x : R, r \rightarrow_p r[x \rightarrow \mu x : R, r] \\
\text{P-MUR} & \quad R \rightarrow_p R' \quad r \rightarrow_p r' \rightarrow_p R' \\
\end{align*}
\]

Fig. 5. One-step decidable parallel reduction of erased terms

\[
\begin{array}{c|c}
|e_1 e_2| & e_1 | e_2 \\
|\lambda x : A \cdot e| & \lambda x : [A], [e] \\
|\Pi x : A, B| & \Pi x : [A], [B] \\
|\mu x : A \cdot e| & \mu x : [A], [e] \\
|\text{cast}_o [A] e| & [e] \\
|\text{cast}_\gamma [A] e| & [e] \\
\end{array}
\]

(Term Erasure)

\[
\begin{array}{c|c}
|\emptyset| & \emptyset \\
|\Gamma, x : A| & |\Gamma|, x : [A] \\
\end{array}
\]

(Context Erasure)

Fig. 6. Erasure of casts

Secondly, the definition of parallel reduction in Fig. 5 is slightly different from the standard one for PTS (Adams, 2006). It is partially parallel: rule P-RED and rule P-MURED do not parallel reduce sub-terms, but only do beta reduction and recursion unrolling, respectively. The confluence property for one-step reduction is lost\(^4\). Nevertheless, such definition makes the decidability property (see Lemma 6.5) easier to prove than the conventional fully parallel version, thus it is called decidable parallel reduction. It also requires fewer reduction steps than the non-parallel version, thus correspondingly needs fewer casts.

Thirdly, parallel reduction does not have the determinacy property like weak-head reduction (Lemma 4.2). For example, for the term \((\lambda x : \text{Int} \cdot 1 + 1)\) 3, we can (parallel) reduce

\[^4\] Notice that multi-step reduction \(\rightarrow_p\) is still confluent since it is equivalent to multi-step full beta reduction \(\rightarrow_p^\ast\) (see Lemma 6.3) which is confluent.
Expressions $r, R ::= x \mid r_1 r_2 \mid \lambda x : R. r \mid \Pi x : R_1, R_2 \mid \mu x : R. r$

Values $u ::= s \mid \lambda x : R. r \mid \Pi x : R_1, R_2$

Contexts $\Delta ::= \emptyset \mid \Delta, x : R$

\[ R_1 \rightarrow R_2 \] 

**(Syntax)**

**(Weak-head Reduction)**

\[
\begin{align*}
\Delta \vdash r : R & \quad \text{(Typing of } \text{PTS}_\mu) \\
\text{TE-AX} & \quad \Delta \vdash (s_1, s_2) \in \sigma \quad \Delta \vdash s_1 : s_2 \\
\quad & \quad \Delta \vdash s_1 \rightarrow s_2 \\
\text{TE-VAR} & \quad \Delta \vdash x : R \in \Delta \\
\quad & \quad \Delta \vdash \lambda x : R \vdash r : R \\
\text{TE-APP} & \quad \Delta \vdash r_1 \vdash \Pi x : R_1, R_2 \\
\quad & \quad \Delta \vdash r_2 \vdash R_1 \vdash r_2[x \mapsto r_2] \\
\quad & \quad \Delta \vdash r_1 r_2 \vdash R_2 \\
\text{TE-PROD} & \quad \Delta \vdash r_1 \vdash R_1 \\
\quad & \quad \Delta, x : R_1 \vdash r_2 : s_2 \\
\quad & \quad (s_1, s_2, s_3) \in \mathcal{S} \\
\quad & \quad \Delta \vdash \lambda x : R_1, R_2 \vdash \Pi x : R_1, R_2 \\
\text{TE-Conv} & \quad \Delta \vdash r_1 \vdash R_1 \\
\quad & \quad \Delta \vdash r_2 \vdash S \\
\quad & \quad \Delta : \equiv \beta R_2 \\
\text{TE-CONS} & \quad \Delta \vdash r : S \\
\quad & \quad x \text{ fresh in } \Gamma \\
\quad & \quad \Delta \vdash \Pi x : R_1, R_2 \\
\quad & \quad \Delta \vdash \Pi x : R_1, R_2 \\
\quad & \quad \Delta \vdash \Pi x : R_1, R_2 \\
\end{align*}
\]

**(Well-formedness)**

\[
\begin{align*}
\vdash \Delta \\
\text{WE-NIL} & \quad \Delta \vdash \emptyset \\
\text{WE-CONS} & \quad \Delta \vdash R : s \\
\quad & \quad x \text{ fresh in } \Gamma \\
\quad & \quad \Delta \vdash \Delta, x : R \\
\end{align*}
\]

Fig. 7. PTS\(_\mu\)

it to either $(\lambda x : \text{Int.} \ 2) \ 3$ by rule P-APP and rule P-ABS, or $1 + 1$ by rule P-RED. Thus, to ensure the decidability, we also need to add the type annotation for cast\(_\beta\) operator to indicate what exact type we want to reduce to. Similarly to cast\(_\beta\), cast\(_\gamma\) $[\Delta] v$ is a value, which is different from the call-by-name/value variant.

Fig. 8 shows the specification of full PITS. Changes from call-by-name/value PITS are highlighted. Note that we do not define operational semantics directly but up to erasure. Reduction relations are defined in PTS\(_\mu\) only for terms after erasure. Similarly, syntactic values are not defined in full PITS but defined for erased terms, ranged over by $u$ in PTS\(_\mu\) (see Figure 7). This is different from call-by-name/value PITS, where reduction rules for type casting and term evaluation are the same, i.e., the one-step call-by-name/value reduction. In full PITS, parallel reduction is only used by casts, while a separate reduction is used for term evaluation. For simplicity reasons, we choose the call-by-name reduction ($\rightarrow$) for term evaluation for erased terms in PTS\(_\mu\) (see Fig. 7).
We show that the two key properties, type safety and decidability of type checking, still hold in full PITS.

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

\[(\text{cast} \downarrow \text{Int} \to \text{Int}) (\lambda x: ((\lambda y: \ast, y) \text{Int}). x)) 3\]

cannot be further reduced because the head position is already a value but not a \(\lambda\)-term. Note that weak casts do not have such problem because only cast, is annotated and it is not legal to have a \(\Pi\)-type in the annotation (see last paragraph of Section 4.5). To avoid getting stuck by full casts, one could introduce several cast push rules similar to System FC (Sulzmann et al., 2007). For example, the stuck term above can be further evaluated by
Then we show Lemma 6.2 holds for one-step full beta reduction, which holds up to the parallel reduction subject reduction and progress theorems for

Notice that term evaluation uses the weak-head reduction

If Theorem 6.1 holds, if

Lemma 6.1 typing of erased terms follow the type system PTS and Guru (Stump et al., 2008) that removes all casts when proving the type safety. The typing of erased terms follow the type system PTS (Barendregt, 1992). The substitution and progress lemmas are stated as follows:

Lemma 6.1 (Substitution of PTS$_{\mu}$)

If $\Delta, x : R', \Delta_1 \vdash r_1 : R$ and $\Delta_1 \vdash r_2 : R'$, then $\Delta, \Delta_2[x \mapsto r_2] \vdash r_1[x \mapsto r_2] : R[x \mapsto r_2]$.

Theorem 6.1 (Progress of PTS$_{\mu}$)

If $\not\vdash r : R$ then either $r$ is a value $u$ or there exists $r'$ such that $r \rightarrow r'$.

Notice that term evaluation uses the weak-head reduction $\rightarrow$. We only need to prove subject reduction and progress theorems for $\rightarrow$. But we generalize the result for subject reduction, which holds up to the parallel reduction $\rightarrow_p$. We first show subject reduction holds for one-step full beta reduction $\rightarrow_{\mu}$ (see Fig. 9) and multi-step full beta reduction $\rightarrow_{\mu}^*$, i.e., reflexive and transitive closure of $\rightarrow_{\mu}$:

Lemma 6.2 (Subject Reduction for Full Beta Reduction)

1. If $\Delta \vdash r : R$ and $r \rightarrow_{\mu} r'$ then $\Delta \vdash r' : R$.
2. If $\Delta \vdash r : R$ and $r \rightarrow_{\mu}^* r'$ then $\Delta \vdash r' : R$.

Then we show $\rightarrow_{\mu}$ is equivalent to multi-step parallel reduction $\rightarrow_{p}$, i.e., transitive closure of $\rightarrow_{p}$ (since it is already reflexive):
Lemma 6.3 (Equivalence of Parallel Reduction)
Given \( r_1 \) and \( r_2 \), \( r_1 \leadsto_p r_2 \) holds if and only if \( r_1 \rightarrow_p r_2 \) holds.

Thus, subject reduction for parallel reduction \( \leadsto_p \) is an immediate corollary:

Theorem 6.2 (Subject Reduction for Parallel Reduction)
If \( \Delta \vdash r : R \) and \( r \rightarrow_p r' \) then \( \Delta \vdash r' : R \).

Finally, given that the PTS\( _p \) is type-safe, if we want to show the type-safety of full PITS, it is sufficient to show the typing is preserved after erasure:

Lemma 6.4 (Soundness of Erasure)
If \( \Gamma \vdash e : A \) then \( [\Gamma] \vdash [e] : [A] \).

Decidability of Type Checking. The proof of decidability of type checking full PITS is similar to call-by-name PITS in Section 4.5. We also limit discussion of decidability to functional PITS (see Definition 4.1). The only difference is for cast rule TF-CASTUP and rule TF-CASTDN, which use parallel reduction \( |A_1| \rightarrow_p |A_2| \) as a premise. We first show the decidability of parallel reduction:

Lemma 6.5 (Decidability of Parallel Reduction)
Given \( r_1 \) and \( r_2 \), it is decidable to determine whether \( r_1 \rightarrow_p r_2 \) holds.

The proof is by induction on the length of \( r_1 \) and attempting to construct a parallel reduction step from \( r_1 \) to \( r_2 \). A checking algorithm can be derived from the proof and runs in linear time in the size of both terms. Notice that the proof does not rely on the single-step confluence of \( \rightarrow_p \). The confluence of \( \rightarrow_p \) is lost due to two base reduction rules P-RED and P-MURED, which do not simultaneously reduce sub-terms but only do one-step beta reduction and recursive unrolling, respectively (see Section 6.1). However, both rules become deterministic, which makes it easier to determine if \( \rightarrow_p \) in both cases. We can just check if one-step reduction of \( r_1 \) is equal to \( r_2 \), similarly to the proof for call-by-name PITS (see Section 4.5). For example, consider \( r_1 = (\lambda x : R_3. r_3) r_4 \) and \( r_2 = r_5 r_6 \). If case P-APP applies, i.e., \( \lambda x : R_3. r_3 \rightarrow_p r_5 \) and \( r_4 \rightarrow_p r_6 \) hold, then \( r_1 \rightarrow_p r_2 \) holds trivially. Otherwise, \( r_1 \rightarrow_p r_2 \) holds if and only if P-RED holds. The latter can be determined by testing the equality of \( r_1[x \mapsto r_4] \) and \( r_5 r_6 \). By contrast, the proof will be much complicated for the standard parallel reduction. To ensure one-step confluence, the rule P-RED becomes

\[
\frac{r_1 \rightarrow_p r_1' \quad r_2 \rightarrow_p r_2'}{\left(\lambda x : R. r_1\right) r_2 \rightarrow_p r_1'[x \mapsto r_2']}
\]

We now need to determine if \( r_1 \rightarrow_p r_1' \) holds or not. This is non-trivial since the induction hypothesis only gives the hint whether \( \lambda x : R_3. r_3 \rightarrow_p r_5 \) holds but not directly for \( r_3 \).

As cast\( _0 \) and cast\( _3 \) are annotated, both \( A_1 \) and \( A_2 \) can be determined and the well-typedness is checked in the original system. By Lemma 6.4, the erased terms keep the well-typedness. By Lemma 6.5, it is decidable to check if \( |A_1| \rightarrow_p |A_2| \). We conclude the decidability of type checking by following lemmas:

Lemma 6.6 (Uniqueness of Typing for functional PITS)
In any functional PITS, if \( \Gamma \vdash e : A_1 \) and \( \Gamma \vdash e : A_2 \), then \( A_1 \equiv A_2 \).
Theorem 6.3 (Decidability of Type Checking for functional PITS)

In any functional PITS, given a well-formed context $\Gamma$ and a term $e$, it is decidable to determine if there exists $A$ such that $\Gamma \vdash e : A$.

### 6.3 Completeness to Pure Type Systems

We have shown that full PITS is complete to a variant of PTS$_{\mu}$. Such variant uses an alternative single-step conversion rule (Geuvers, 1995):

\[
\frac{\Delta \vdash r : R_1 \quad \Delta \vdash R_2 : s \quad R_2 \rightarrow^\beta R_1}{\Delta \vdash r : R_2}
\]

\[
\frac{\Delta \vdash r : R_1 \quad \Delta \vdash R_2 : s \quad R_1 \rightarrow^\beta R_2}{\Delta \vdash r : R_2}
\]

where $\rightarrow^\beta$ denotes full beta reduction (see Fig. 9). We call this variant PTS$_{\text{step}}$ (see Fig. 10). Its typing judgment is denoted by $\Delta \vdash: R$. PITS can be seen as an annotated version of PTS$_{\text{step}}$. We first show that full PITS is complete to PTS$_{\text{step}}$ by the following lemma:
Lemma 6.7 (Completeness of Full PITS to PTS<sub>step</sub>)
If $\Delta \vdash r : R$, then there exists $\Gamma, e$ and $A$ such that $\Gamma \vdash e : A$ where $|\Gamma| = \Delta$, $|e| = r$ and $|A| = R$.

Furthermore, Siles and Herbelin have proved that single-step conversion rule is equivalent to the original conversion rule using beta conversion in PTS (see Corollary 2.9 in (Siles & Herbelin, 2012)). We have the following relation between PTS<sub>step</sub> and PTS<sub>µ</sub>:

Lemma 6.8 (Completeness of One-step PTS to PTS)
If $\Delta \vdash r : R$, then $\Delta \vdash r : R$ holds.

Thus, we can conclude the full PITS is complete to PTS<sub>µ</sub>:

Theorem 6.4 (Completeness of Full PITS to PTS<sub>µ</sub>)
If $\Delta \vdash r : R$, then there exists $\Gamma, e, A$ such that $\Gamma \vdash e : A$ where $|\Gamma| = \Delta$, $|e| = r$ and $|A| = R$.

7 Discussion and Related Work

In this work we have developed three variants of PITS. The three variants differ on the reduction rules used in the cast operators. As it turns out the specific choice of reduction has quite a bit of impact on the formalization and the metatheory. In this section we start by presenting an extensive comparison between the three variants of PITS, as well as PTS<sub>f</sub> (which is a closely related variant of PTS). Then we discuss other closely related work.

7.1 Comparing the three variants of PITS and PTS<sub>f</sub>
We have developed PITS with the aim of using such family of calculi as foundations to programming languages supporting unified syntax and recursion. PITS trades the convenience of implicit type conversion that is afforded in most dependently typed calculi by a simple meta-theory that allows for decidable type-checking. Closely related to our work is PTS with explicit convertibility proofs (PTS<sub>f</sub>) (van Doorn et al., 2013), which is a variant of PTS. PTS<sub>f</sub> replaces the conversion rule by embedding explicit conversion steps into terms. PTS<sub>f</sub> has strong connections to PITS in the sense that both systems are based on Pure Type Systems and require explicit type conversions — PTS<sub>f</sub> uses proof-annotated terms, while PITS uses cast operators. Nevertheless the design of PTS<sub>f</sub> is still quite complex, as it requires a separate sub-language for building convertibility (equality) proofs.

Although PTS<sub>f</sub> was motivated by applications to theorem proving, PTS<sub>f</sub> (like PTS) can be instantiated to form inconsistent calculi, which can encode fixpoints and general recursion. Therefore PTS<sub>f</sub> could also, in principle, be used as a foundation for programming languages. However, the underlying mechanisms and foundations of PTS<sub>f</sub> and PITS are different. Generally speaking, when dealing with programming languages with general recursion it is important to study not only the static semantics, but also the dynamic semantics. Unlike strongly normalizing languages where any choice of reduction leads to termination, in languages that are not strongly normalizing this is not usually the case and the choice of reduction is important. Furthermore the choice of the style of reduction in casts has a profound impact on the properties and metatheory of the language. Our
Table 2. Comparison between PTS\(_f\) and PITS

<table>
<thead>
<tr>
<th>Features</th>
<th>PTS(_f)</th>
<th>Call-by-name PITS</th>
<th>Call-by-value PITS</th>
<th>Full PITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Dynamic Semantics</td>
<td>(\bigcirc)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bigcirc)</td>
</tr>
<tr>
<td>Direct Proofs</td>
<td>(\bigcirc)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bigcirc)</td>
</tr>
<tr>
<td>No Mutually Dependent Judgments</td>
<td>(\bigcirc)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bigcirc)</td>
</tr>
<tr>
<td>Implicit Proofs by Reduction</td>
<td>(\bigcirc)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
</tr>
<tr>
<td>Full Type-Level Computation</td>
<td>(\bullet)</td>
<td>(\bigcirc)</td>
<td>(\bigcirc)</td>
<td>(\bullet)</td>
</tr>
<tr>
<td>Consistency of Reduction</td>
<td>(\bigcirc)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bigcirc)</td>
</tr>
<tr>
<td>Decidability with Recursion</td>
<td>(\bigcirc)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
</tr>
<tr>
<td>Erasability of Casts</td>
<td>(\bigcirc)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
</tr>
<tr>
<td>Equality</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
</tr>
<tr>
<td>No Value Restriction</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bigcirc)</td>
<td>(\bullet)</td>
</tr>
<tr>
<td>SLOC of Coq proofs</td>
<td>7318</td>
<td>1217</td>
<td>1477</td>
<td>3796</td>
</tr>
<tr>
<td>Lemmas of Coq proofs</td>
<td>319</td>
<td>62</td>
<td>66</td>
<td>221</td>
</tr>
</tbody>
</table>

1 Proofs for typing decidability are direct, but not for type-safety.
2 We believe that decidability should hold, but no there is no discussion or proofs in the formalization of PTS\(_f\) (van Doorn et al., 2013).
3 Encoding via Leibniz equality.

work on PITS puts great emphasis on the study of the dynamic semantics and the trade-offs between different choices, while in PTS\(_f\) only the static semantics is studied. Next we give a detailed comparison on features between PTS\(_f\) and the three variants of PITS, summarized in Table 2.

**Direct Dynamic Semantics.** One important difference between call-by-value and call-by-name PITS and the variant with full reduction is that the former two calculi have a direct small-step operational semantics, while the semantics of the later calculus is indirectly given by elaboration. A direct operational semantics has the advantage that the reduction rules can be used to directly reason about expressions in the calculus. This reasoning can be used to perform, for example, equational reasoning steps or to justify the correctness of some optimizations. In an elaboration-based semantics the lack of reduction rules means that one must first translate the source expression into a corresponding expression in the target calculus, and then do all reasoning there. This is a much more involved process.

As discussed in Section 6.2, it is difficult to directly define a type-preserving operational semantics for full PITS. The problem is not intrinsic to PITS, but rather it is a general problem whenever full reduction is used in cast-like operators. Indeed this problem has been identified previously in the literature (Sjöberg et al., 2012; Sulzmann et al., 2007), and two major approaches have been used to address it. One approach is not to use a direct semantics but instead to use an elaboration semantics, which is precisely the approach that we used in our variant of PITS with full reduction. This approach is quite common and it is also the approach used in PTS\(_f\) as well as several other calculi (Sjöberg et al., 2012;
Pure Iso-Type Systems

Sjöberg & Weirich, 2015; Stump et al., 2008). Another approach that has been presented in the literature is to use push rules as in System FC (Sulzmann et al., 2007; Yorgey et al., 2012; Weirich et al., 2013) and System DC (Weirich et al., 2017). However push rules significantly complicate the reduction rules (see Section 6.2).

In this work we show a third approach to achieve a simple type-preserving direct dynamic semantics: we can use alternative weaker reduction relations (call-by-value or call-by-name) for type conversion. The weaker reduction relations are straightforward and do not have the extra complication of the push rules (although some expressiveness is lost).

There is no discussion on how to achieve direct dynamic semantics using the proof term approach by PTS, since they use an elaboration approach. Furthermore this is unlikely to be trivial. We expect that it may be possible to give a direct operational semantics to full PITS or PTS using push rules similar to the ones employed in System DC (Weirich et al., 2017), but this would come at the cost of a much more involved set of reduction rules (as well as the corresponding meta-theory).

**Direct Proofs.** Because of the direct dynamic semantics it is possible to do direct proofs of preservation and progress in call-by-value and call-by-name PITS. The meta-theory of Full PITS is significantly more involved because we need a target calculus and to prove several lemmas in both the target and the source, as well as showing the correspondence between the two systems. To give a rough idea of the complexity of the different developments, Table 2 shows the total number of lines of Coq code used and lemmas used to formalize the three variants of PITS. Roughly speaking the development of full PITS requires twice as many SLOC and nearly four times more lemmas than the other two variants, since we also need to formalize PTS along with full PITS.

PTS is shown to be equivalent to plain PTS and its type-safety then can be guaranteed by showing the correspondence to plain PTS (Barendregt, 1991). However the proof for soundness and completeness between PTS and PTS is highly non-trivial (van Doorn et al., 2013; Siles & Herbelin, 2012). In PTS there is no discussion on proving subject reduction directly in PTS. The type-safety of PTS is indirectly shown by erasure of explicit proofs to generate valid plain PTS terms. This is similar to the proof strategy for type-safety of full PITS, which is shown up to erasure of casts (see Section 6.2). The formalization of PTS requires about 7000 SLOC and 300 lemmas (see Table 2), including auxiliary systems such as plain PTS and PTS. These numbers cannot be directly compared to the numbers of the PITS formalizations, since different approaches and libraries are employed to deal with binding and the formalization of PTS does not include proofs of decidability of type-checking. Nevertheless the numbers are useful to give an idea of the effort in the PTS formalization.

Ultimately we believe that direct proofs and a direct operational semantics of the call-by-name and call-by-value PITS are quite simple. Furthermore such simplicity is helpful when trying to extend calculi to study additional features. One non-trivial extension that we have already studied, based on a particular instance of the call-by-name PITS, is subtyping (Yang & Oliveira, 2017). Integrating subtyping and some form of dependent types is a widely acknowledged difficult problem (Aspinall & Compagnoni, 1996; Castagna & Chen, 2001; Hutchins, 2010). Nevertheless, using our call-by-name instance of PITS extension with subtyping we have managed to develop a calculus that subsumes System
No Mutually Dependent Judgments. \(\text{PTS}_f\) requires more language mechanisms for type conversions, including proof terms \((H)\) and their typing rules \((\Gamma \vdash_f H : A = B)\) to ensure coercions \((A = B)\) are well-typed. However, the well-formedness checking of coercions depends on typing judgments \((\Gamma \vdash_f e : A)\), which causes mutual dependency of judgments and complicates proofs. Casts in PITS use reduction relations \((A \rightarrow B)\), which are untyped and do not depend on typing rules. The well-formedness of types is checked separately in typing rules of cast operators, e.g. TF-CASTUP and TF-CASTDN in full PITS. The fact that PITS does not require such mutually dependent judgments means that many proofs can be proved using simple inductions. In \(\text{PTS}_f\) the mutually dependent judgments leads to several lemmas that need to be mutually proved.

Implicit Proofs by Reduction. \(\text{PTS}_f\) uses coercions (i.e. equivalence relations) to explicitly write equality proofs, while PITS uses reduction relations that implicitly construct such proofs. Equality proofs in \(\text{PTS}_f\) are constructed by proof terms. Each language construct requires a corresponding proof term to reason about equality of sub-terms, which adds to the number of language constructs. On the contrary, PITS does not require proof terms but two extra cast operators, thus has fewer language constructs.

Type conversions in \(\text{PTS}_f\) are more “explicit” than in PITS. One needs to specify which proof terms to be used exactly in \(\text{PTS}_f\), while he/she just needs casts without specifying which underlying reduction rule to use. This makes it easier to do explicit type conversions in PITS. Assume that we have integer literals and use beta reduction to evaluate addition \((1 + 1 \rightarrow 2)\). Recall the example from Section 1:

\[
e : \Pi x : \text{Int. Vec} (1 + 1)
e' : \Pi x : \text{Int. Vec} 2
\]

To obtain \(e'\) from \(e\), in \(\text{PTS}_f\), \(e' = e^H\) where \(H\) is a proof term such that

\[
H = \{\text{Int}, [x : \text{Int}](\text{Vec} \beta (1 + 1))\}
\]

In full PITS, \(e' = \text{cast}_\beta [\Pi x : \text{Int. Vec} 2] e\), which implicitly uses the reduction rules P-PROD, P-APP and P-RED in the cast operator. In call-by-name/value PITS, due to the determinacy of reduction used in casts, the \(\text{cast}_\beta\) operator even does not require a type annotation. For example, consider a simpler type conversion \((\lambda x : *. x) \text{Int} \rightarrow \text{Int}\) from Section 2.3:

\[
e : (\lambda x : *. x) \text{Int}
e' : \text{Int}
\]

where \(e'\) can simply be \(\text{cast}_\beta e\) without any annotation.

Generally speaking we believe that for programming, and especially more traditional forms of programming that do not involve complex forms of type-level reasoning, having such implicit proofs of conversion is good. We believe that for a practical language design to be based directly on \(\text{PTS}_f\) it would require some degree of inference of the equality proof terms. In some sense full PITS can be thought of a system that implicitly generates
proof terms and in principle could be translated to $\text{PTS}_f$, but it is one step closer to a source language that infers equality proof terms. Of course if the goal is to do theorem proving and/or heavy uses of type level computation then having explicit control over the equality proof terms can be an advantage. However for PITS our focus is on programming languages with general recursion.

**Full Type-Level Computation.** $\text{PTS}_f$ has full type-level computation since type conversion uses equivalence relations which are congruent. It has the same expressive power as Pure Type Systems. Full PITS similarly has full type-level computation, while call-by-name/value PITS do not. Full type-level computation is useful for theorem proving and full-spectrum dependently typed programming as in Coq and Agda, but not necessarily required for traditional programming. As the examples presented in Section 3 show, iso-types that just use weak-head call-by-name reductions in casts can encode many advanced type-level features. The extra expressiveness of $\text{PTS}_f$ and full PITS also comes at the cost of additional complexity in the metatheory and makes it non-trivial to achieve features like direct dynamic semantics and direct type-safety proofs.

**Consistency of Reduction.** In many strongly normalizing languages a basic assumption is that the order of reduction does not matter. This justifies reasoning that can be done in any order of reduction. Reduction strategies such as parallel reduction embody this principle and enable reductions in terms to occur in multiple orders. However, in languages with general recursion this assumption is broken: i.e. reduction order does matter. For example, given term $(\lambda x : \text{Int}. \, 1) \downarrow$, where $\downarrow$ is any diverging computation, such term evaluates to $1$ with call-by-name semantics, but diverges with call-by-value semantics. If we want to conduct precise reasoning about programs and their behavior we cannot ignore the order of reduction. In particular, if we want the type-level reduction to respect the runtime semantics/reduction then we need to ensure that the two reductions are in some sense consistent.

We study three different variants of PITS that differ on term evaluation strategy, as well as the reduction strategy employed by the cast operators. They have different trade-offs in terms of simplicity and expressiveness. Call-by-name PITS uses weak-head call-by-name reduction, while call-by-value PITS enables standard call-by-value reduction by employing a value restriction (Swamy et al., 2011; Sjöberg et al., 2012). In both designs the key idea is that the same reduction strategy is used for both term evaluation and type casts, ensuring a consistent behavior of reduction at both type and term level. In call-by-name/value PITS, we can trivially guarantee that no invalid reasoning steps can happen due to mismatches with the evaluation strategy.

In full PITS, the use of parallel reduction breaks consistency because type-level reduction allows some reductions that are not allowed at the term-level. For example, $\text{Int} \rightarrow \text{Vec} (1 + 1)$ can be reduced to $\text{Int} \rightarrow \text{Vec} 2$ by type-level parallel reduction but not term-level reduction. We believe that it may be possible to have a variant of PITS that uses call-by-name or call-by-value and has a consistent form of full reduction that respects the reduction order. However we leave this for future work.

$\text{PTS}_f$ has no discussion on reduction rules for term evaluation, since its dynamic semantics is given by elaboration into PITS. Since the focus of $\text{PTS}_f$ is primarily on the
applications to theorem proving the issues of consistency between term and type-level reduction are not relevant, because such for theorem proving calculi are normally strongly normalizing and reduction order does not affect the semantics.

Decidability in the Presence of Recursion. There is no formal discussion or direct proof on decidability of the type system for PTS\(_f\), though this seems to be a plausible property since the typing rules of PTS\(_f\) are syntax-directed. Only uniqueness of typing is formally discussed and proved for functional PTS\(_f\). This is similar to functional PITS which have uniqueness of typing up to only alpha equality due to the absence of implicit beta conversion (see Lemma 4.1). Uniqueness of typing is used in the decidability proof of functional PITS (see Section 4.5), and we believe that it should be useful to prove decidability of PTS\(_f\) as well.

Alternatively, note that an indirect proof for decidability of typing can be derived from the plain metatheory of PTS through the equivalence of PTS\(_f\) and PTS. However, the original decidability proof for PTS relies on the normalization property (van Benthem Jutting, 1993). Thus, non-normalizing PTS\(_f\) cannot use such indirect approach to prove decidability for variants of PTS\(_f\) with recursion/fixpoints.

For all three variants of PITS, decidability of type checking has been proved directly in the presence of general recursion without relying on normalization, though the proof is done only for functional PITS for simplicity reasons. We expect that a similar proof would work for PTS\(_f\) as well, and this would be interesting to prove in future work.

Erasability of Casts. Type casts are usually erasable after type checking to avoid run-time cost. Erasure of casts is only valid if they are computationally irrelevant, meaning that the operational semantics of expressions does not change after erasure. As mentioned in Section 6.1, casts in call-by-value and full PITS are erasable. In full (call-by-value) PITS, \(\text{cast}_V [A] \ v \ (\text{cast}_V [A] \ v)\) is a value and becomes \(v\) after erasure, which is still a value and does not change the computational behavior. Similarly, equality proofs in PTS\(_f\) are also computationally irrelevant and erasable. In contrast, casts in call-by-name PITS do not have this property, as we define \(\text{cast}_N [A] \ e\) to be a value. If we erase the cast operator, \(e\) can be further reducible, which changes the dynamic semantics. Nonetheless, it is easy to gain erasability of casts in call-by-name PITS by partially adopting the call-by-value rules for cast\(_V\) specifically. For example, we can define \(\text{cast}_V [A] \ v\) as a value and further reduce inner terms of a cast\(_V\) by rule RV-CASTUP (see Fig. 4). We did not adopt this approach in call-by-name PITS, since such operational semantics requires additional reduction rules and complicates the metatheory.

Equality. PTS\(_f\) has built-in support for propositional equality, while PITS needs to encode equality types using Leibniz equality. In Section 3.4, we show the encoding of Leibniz equality, which works for all three variants of PITS. But full PITS is required for complete reasoning on equality types. In call-by-name/value PITS, equality reasoning is limited since type conversions are not congruent. The reasoning required for vectors (e.g., tail function in Section 3.4) is not encodable.
Pure Iso-Type Systems

No Value Restriction. We employ a value restriction in call-by-value PITS to retain subject reduction with a simple proof. As discussed in Section 5.1, we will consider an alternative approach by adding cast operators during dynamic semantics for future work. In contrast, call-by-name and full PITS, as well as PTS_f, do not rely on the value restriction to prove subject reduction.

7.2 Other Related Work

Core calculus for functional languages. Girard’s System F_o (Girard, 1972) is a typed lambda calculus with higher-kindred polymorphism. For the well-formedness of type expressions, an extra level of kinds is added to the system. In comparison, because of unified syntax, PITS is considerably simpler than System F_o, both in terms of language constructs and complexity of proofs. As for type-level computation, System F_o differs from PITS in that it uses a conversion rule, while PITS uses explicit casts. PITS is also inspired by the treatment of datatype constructors in Haskell (Jones, 1993). Iso-types have similarities to newtypes and datatypes which involve explicit type-level computations. The current core language for GHC Haskell, System FC (Sulzmann et al., 2007) is a significant extension of System F, which supports GADTs (Peyton Jones et al., 2004), functional dependencies (Jones, 2000), type families (Eisenberg et al., 2014), and kind equality (Weirich et al., 2013). System DC (Weirich et al., 2017) is a further extension to FC and foundation of Dependently Typed Haskell (DTH) that extends Haskell with full-spectrum dependent types. System DC is not logically consistent and can support recursion as well as decidable type-checking. However System DC is still quite ambitious in that it comes with a rich notion of type equality aimed at supporting more sophisticated forms of type-level computation. While such rich notion of type equality adds expressive power, the details of type equality are quite involved and System DC is still significantly more complex than classic pure type systems or older Haskell/ML-style language designs based on System F (Reynolds, 1974; Girard, 1972).

System FC and its extensions require a non-trivial form of type equality, which is currently missing from PITS. One possible direction for future work is to investigate the addition of such forms of non-trivial type-equality. On the other hand, PITS uses unified syntax and has only 8 language constructs, whereas the original System FC uses multiple levels of syntax and currently has over 30 language constructs, making it significantly more complex. The simplicity of PITS makes it suitable to be combined with other language features, such as subtyping, which seems hard to support in FC and its extensions. For example, we proposed $\lambda I_c$ (Yang & Oliveira, 2017), in a separate article, which extends a variant of PITS called $\lambda I$ with bounded quantification and subtyping and supports object encodings (Pierce & Turner, 1994).

Unified syntax with decidable type-checking. Pure Type Systems (Barendregt, 1991) 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. An early attempt of using a PTS-like syntax for an intermediate language for functional programming was Henk (Peyton Jones & Meijer, 1997). 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, full dependent types or the meta-theory.

Zombie (Casinghino et al., 2014) 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 (Sjöberg et al., 2012; Sjöberg & Weirich, 2015). In contrast to Zombie, PITS 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 (Augustsson, 1998) integrates the full power of dependent types with general recursion, which bears some similarities with PITS. It uses one syntactic form for both terms and types, allows arbitrary computation at type level and is logically inconsistent because of the presence of unrestricted recursion. Moreover, the most crucial difference from PITS 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 (Cardelli, 1986) 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. ΠΣ (Altenkirch et al., 2010) 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, ΠΣ 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 PITS is controlled by explicit casts. Several studies (Stump et al., 2008; Sjöberg et al., 2012; Kimmell et al., 2012; Sjöberg & Weirich, 2015; Sulzmann et al., 2007; Gundry, 2013; Weirich et al., 2017) 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 PITS 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 (Stump et al., 2008) and Sep3 (Kimmell et al., 2012) make syntactic separation between proofs and programs to prevent certain programmatic terms turning into invalid proofs. System DC (Weirich et al., 2017) and other FC variants (Sulzmann et al., 2007; Yorgey et al., 2012) similarly distinguish coercions (i.e. equality proofs) and programs syntactically. The programmatic part of Zombie (Sjöberg et al., 2012; Sjöberg & Weirich, 2015), which has no such separation, employs value restriction that restricts proofs to be syntactic values to avoid non-terminating terms. Gundry’s evidence language (Gundry, 2013) also unifies all
syntactic levels including coercions, but uses different phases for separating programs and proofs. The typing rules contain an access policy relation to determine the conversion of phases. Such mechanism is finer-grained yet more complicated. Note that our treatment of full casts in full PITS, 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 (The Coq development team, 2016) and Adga (Norell, 2007) are conservative as to what kind of computation is allowed. They require all programs to terminate by means of a termination checker, ensuring that 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 styles. 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 (Brady, 2011) 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 PITS. Instead PITS 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 and the meta-theory of the core language.

8 Conclusion and Future Work

This work proposes PITS: a family of minimal dependently typed core languages that allow the same syntax for terms and types, support type-level computation, and preserve 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. Different variants of PITS show trade-offs of employing different reduction strategies in casts.

In future work, we hope to investigate more applications of iso-types. Two possible directions are discussed next:

**Type-level computation and subtyping.** One possible direction is to implement more surface language mechanisms, so as to express intensive type-level computation in a more convenient way. For example, we can add lightweight (but not full) type inference for casts, or multi-step casts with step limits. Another direction is to combine iso-types with more language features, e.g., subtyping and strong dependent sums, which are useful to model object-oriented programming constructs and to model module systems. Thanks to the simplicity of PITS, we can extend PITS without complicating the metatheory too much, as our previous work on \(\lambda L_c\) (Yang & Oliveira, 2017) illustrates. Such extensions of PITS could also be used as underlying theories of feature-rich languages, similarly to Dependent Object Types (DOT) of Scala (Amin et al., 2012). We plan to develop a language based on iso-types and unified subtyping (Yang & Oliveira, 2017), which will have similar features.
to DOT and be able to model Scala-like programming idioms, e.g., type members and path-dependent types.

**Traditional language designs and type-inference.** It will be also interesting to use PITS to model traditional functional languages like (older versions of) Haskell and ML with some extra features that come “for free” from the use of dependent types and unified syntax. We believe PITS should easily allow the development of functional languages that stand somewhere in-between older designs of functional languages (such as Haskell 98 and the original ML) and dependently typed languages like Idris (Brady, 2011) and Agda (Norell, 2007) or dependently typed Haskell (Weirich et al., 2017). One short term goal for the future is to study a version of the **Fun** language with type-inference.

We believe that the absence of a conversion rule makes the problem of type-inference and unification considerably easier than for dependently typed languages with a conversion rule. Type-inference is still a major difference between ML-like languages (and Haskell) and existing dependently typed languages. ML-like languages follow the Hindley-Milner (Hindley, 1969; Milner, 1978; Damas & Milner, 1982) tradition and support global type-inference requiring very few or no type-annotations at all. Part of the reason why type-inference works so well in those languages is that only first-order unification is necessary. In contrast dependently typed languages, like Agda or Idris, typically employ local type inference algorithms that require a considerable number of type annotations. This is needed because the unification problem in such dependently typed languages is very hard. Essentially the presence of type level functions and a conversion rule means that a lot of types of different syntactic forms are equivalent. In the general case this would require higher-order unification, which is undecidable (Huet, 1973). Because there is no conversion rule in PITS the unification problem should be considerably simpler as two types are only equivalent if they have the same syntactic form. If the unification problem for PITS (or representative subsystems of PITS) is solved, this should enable forms of type-inference closer to Haskell or ML.

**Acknowledgements**

We would like to thank the anonymous reviewers for their insightful and useful comments. This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17258816.

**References**


Y. Yang and B. Oliveira


