Abstract of thesis entitled

"**Iso-Type Systems: Simple Dependent Type Theories for Programming**"

Submitted by

Yanpeng Yang

for the degree of Doctor of Philosophy
at The University of Hong Kong
in January, 2019

Dependent types have been drawing a lot of attention in programming language designs. A key reason is that they allow unifying types and terms that are usually distinct syntactic levels in traditional language designs. Unified syntax brings some interesting advantages over separate syntax, including added expressiveness and less duplication of concepts. However, it is challenging to combine dependent types with common programming features, such as unrestricted general recursion and object-oriented programming (OOP) features including subtype polymorphism and abstract type members. To address these challenges, we propose three novel dependently typed calculi with both simplicity and expressiveness, namely Pure Iso-Type Systems (PITS), the $\lambda I_{\leqslant}$ calculus and the λI_{Σ} calculus.

PITS is a generic language framework that employs unified syntax, supports general recursion and preserves decidable type checking. It is comparable in simplicity to pure type systems (PTS), and is useful to serve as a foundation for functional languages that stand in-between traditional ML-like languages and full-spectrum dependently typed languages. The key to retain decidable type checking in the presence of general recursion is a generalization of iso-recursive types called iso-types. Iso-types replace the implicit conversion rule typically used in dependently typed calculi and make every computation explicit via cast operators. We study three variants of PITS that differ on the reduction strategy employed by the cast operators and prove type-safety and decidability of type checking for all variants.

The $\lambda I_{\leqslant}$ calculus is a variant of PITS with unified subtyping, a novel technique that unifies typing and subtyping and enables the combination of dependent types and subtyping. In $\lambda I_{\leqslant}$, there is only one judgment that is akin to a typed version of subtyping. Both the typing relation, as well as type well-formedness are just special cases of the unified subtyping relation. $\lambda I_{\leqslant}$ supports essential features for modeling OOP, such as high-order subtyping, bounded quantification and top types. It can fully subsume System $F_{\leqslant}$ and enjoys several standard and desirable properties, such as type-safety and transitivity of subtyping.

The λI_{Σ} calculus is a variant of $\lambda I_{\leqslant}$ with strong dependent sums. Strong sums are useful to model Scala-like traits with type members. λI_{Σ} adopts a novel treatment of strong sums called iso-strong sums. The destructors of iso-strong sums are typed using intermediate type-level applications instead of standard direct substitutions. The necessary type-level computation can be done by just call-by-value casts. λI_{Σ} supports impredicativity and enjoys the same desirable properties as $\lambda I_{\leqslant}$, including type-safety and transitivity. (421 words)

Iso-Type Systems: Simple Dependent Type Theories for Programming

Yanpeng Yang
楊彥芃

Department of Computer Science
The University of Hong Kong

This dissertation is submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

January, 2019

Declaration

I hereby declare that the thesis entitled "*Iso-Type Systems: Simple Dependent Type Theories for Programming*" represents my own work and has not been previously submitted to this or any other institution for a degree, diploma or other qualifications.

Yanpeng Yang
January, 2019

Acknowledgements

My thanks first go to my supervisor, Bruno Oliveira, who has been a great research advisor and mentor. Bruno is so energetic and positive on research. Every meeting with him is just a pleasure. He is very good at guiding students with his great patience and strong expertise. I really appreciate the time he spent on brainstorming new research ideas with me and offering constructive suggestions to me. I am very grateful to all his help and support.

During my time as a PhD student HKU, I am very fortunate to meet so many nice colleagues and friends. The Programming Languages Group is a great place to do research and share laughter. I would like to thank everyone in the group: Haoyuan Zhang, Weixin Zhang, Yanlin Wang, Zhiyuan Shi, Xuan Bi, Ningning Xie, Huang Li, Jinxu Zhao, Xuejing Huang, Tomas Tauber, João Alpuim and other people, just to name a few. I would also like to thank visiting scholars to our group, who patiently listened to my research work and gave me helpful feedback: Prof. Tom Schrijvers, Dr. Marco Servetto, Dr. Vilhelm Sjöberg, Dr. Tomoyuki Aotani (we also met at OOPSLA'17), Dr. Nicholas Ng and Dr. Shin-Cheng Mu (who is also my thesis examiner).

The Department of Computer Science at HKU is a warm place to stay. I would like to thank many professors at the department: my second supervisor Dr. Reynold C. K. Cheng, former Heads of Department Prof. Francis C. M. Lau and Prof. Wenping Wang, Dr. Heming Cui, Dr. Chuan Wu, Dr. Zhiyi Huang and Dr. Hubert Chan. I would also like to thank staffs from the general office and technical support who are kind and always ready to help: Priscilla Chung, Olive Hui, Veronica Yim, Patrick Au and the current and former clean ladies (sorry I forgot your names) who helped me clean my office every week.

HKU is a fantastic place with so many brilliant and kind people. I have met many new friends here who are really kind and helpful. My special thanks go to Luwei Cheng and Dominic Hung Chit-Ho for your encouragement that helped me get over my most difficult time at HKU. I would like to thank Jiancheng Ye from EEE department for showing me around Hong Kong when I was new here. I would like to thank Kan Wu for your professional skills as Soldier: 76 and Mei for covering me. I would like to thank colleagues from the Systems Lab: Huanxin Lin, Hao Wu, Mingzhe Zhang, Weizhi Liu, King Tin Lam and the visiting professor Dr. Dengke Zhang, for the past time when we worked together. I would like to thank other friends I met at HKU and I really enjoy the happy time with you: Zhangquan Wu, Fangying Wang, Shirui Lu, Yong Xu, Gongxian Zeng, Lingjie Liu, Jiatao Gu, Kai Han, Wei Liu, Haofeng Li, Qihang Sun, Yongfei Wang, Yujie Cao, Zhou Xu, Shuang Tong and many names I forgot to mention.

I would also like to thank my old friends, who did not forget me when I was deserted in Hong Kong. I would like to thank my undergraduate roommates and friends, who are just like brothers: Sheng'guo Ren, Fan Gao, Yuxuan Shui and Yuzui Ye. I would like to thank my high school classmates who welcomed me when I first came to Hong Kong, who gathered with me at Hong Kong and who still remembered me and contacted me when I was in Hong Kong: Jiayan

Jiang, Tianyi Yang, Yushang Tang, Yumeng Guo, Xiaoyu Chen, Feng Zhu, Yue Shen, Chen Hang, Xiaozhuo Cheng, Yuxin Chen, Peilu Wang and so many others.

I would like to thank my parents and my family. You are always my powerful and secure backing.

Finally, I especially offer my grateful thanks to my girlfriend Huoru Zhang for your constant support throughout my PhD study. I could never have done this work without your love and encouragement.

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 Type Features of Static Type Systems

Type systems describe how types are assigned for terms in programming languages. Static type checking employs rules of type systems to verify types at compile-time. Static type systems help detect errors early before the program actually runs. For example, if the operator "`/`" performs integer division and one accidentally writes a division by two on the string, i.e. `"3"/2`, the compiler will detect a type mismatch that the operand `"3"` is not an integer. Such ill-typed term will be rejected by the compiler and prevented from crashing at run-time. From the perspective of engineering, static type systems can be beneficial for code refactoring and optimization by utilizing the extra typing information about programs. There is even a trend of adding static type systems to dynamically typed languages. For example, Python allows type annotations starting from Python 3.5 [van Rossum et al. 2014] and TypeScript [Microsoft Corporation 2016] brings an optional type system to JavaScript through gradual typing [Siek and Taha 2006].

Subtyping. Statically typed languages support various type features through their type systems. For example, object-oriented programming (OOP) languages, such as Java [Gosling et al. 1996] and Scala [Odersky et al. 2004], usually support *subtype polymorphism* of objects, which is a key feature of OOP. Such mechanism relies on the *subtyping* relation of type systems. In Scala, we can define classes that implement the same method but behave differently:

```
trait Shape {
    def area(): Double
}

class Square extends Shape {
    var width = 1.0
    def area() = width * width
}

class Circle extends Shape {
    var radius = 1.0
    def area() = Math.PI * radius * radius
}
```

Both `Square` and `Circle` classes are *subtypes* of `Shape`, meaning that we can pass `Square` or

`Circle` objects to the places expecting `Shape` objects. For example, we can define a generic function `PrintArea` that prints the area of a `Shape` object. We can pass a `Square` or `Circle` object to `PrintArea` and get different results:

```
def PrintArea(s: Shape) = println(s.area())
PrintArea(new Square())     // 1.0
PrintArea(new Circle())     // 3.141592653589793
```

Higher-kinded Polymorphism. In contrast to the subtype polymorphism in OOP languages, type systems of functional programming languages, such as Haskell [Marlow et al. 2010] and ML [Milner et al. 1990], usually adopt other forms of polymorphism. For example, Haskell supports *higher-kinded polymorphism*, which is achieved by *higher-kinded types* that can take other types and construct a new type. One example are *monads*, a useful design pattern that can represent sequential computations [Wadler 1995]. Monads in Haskell are defined by a *typeclass* [Hall et al. 1994] (like an interface in Java) with two functions, bind ($\gg=$) and $return$:

$$
\begin{aligned}
&\textbf{class } Monad\ m \textbf{ where}\\
&\quad (\gg=) :: m\ a \to (a \to m\ b) \to m\ b\\
&\quad return :: a \to m\ a
\end{aligned}
$$

$Monad$ is an example of a *higher-kinded* type, which has the following kind:

$$Monad :: (\star \to \star) \to Constraint$$

It takes a type constructor m of kind $\star \to \star$ and returns a typeclass constraint. This enables *abstraction over not only types but also type constructors*, thus obtaining higher-kinded polymorphism. For example, many type constructors in Haskell can be categorized into monads and operated using the bind or return functions, such as list types and $Maybe$ types (also known as *option types*). With higher-kinded polymorphism, it is possible to write a generic $fmap$ function that performs operations for each element inside a monad:

$$
\begin{aligned}
&fmap :: Monad\ m \Rightarrow (a \to b) \to m\ a \to m\ b\\
&fmap\ f\ x = x \gg= \lambda y \to return\ (f\ y)
\end{aligned}
$$

The type constructor m can be a list or option type, which are both monads. For example, we can apply $fmap\ (+1)$ to a list or option to increase contained elements by one:

$$
\begin{aligned}
&fmap\ (+1)\ [1,2,3] \quad \text{-- returns [2, 3, 4]}\\
&fmap\ (+1)\ (Just\ 2) \quad \text{-- returns (Just 3)}
\end{aligned}
$$

Dependent Types. Besides traditional OOP and functional languages, several new programming languages, notably Agda [Norell 2007a] and Idris [Brady 2013], are emerging and employ a different programming paradigm called *dependently typed programming*. Type systems in these languages support *dependent types*: types that can depend on terms. For example in Idris, we can define a *vector*, a special list whose type carries its length:

$$
\begin{aligned}
&\textbf{data } Vec : Nat \to \mathsf{Type} \textbf{ where}\\
&\quad Nil : Vec\ 0\\
&\quad (::) : Int \to Vec\ n \to Vec\ (1+n)
\end{aligned}
$$

The vector type Vec is a dependent type that depends on a natural number Nat, i.e., the length of the list. Dependent types involve *type-level computation*, i.e., evaluating terms in types. For example, consider an $append$ function that concatenates two vectors:

$$
\begin{array}{l}
append : Vec\ n \rightarrow Vec\ m \rightarrow Vec\ (n + m) \\
append\ Nil \qquad ys = ys \\
append\ (x :: xs)\ ys = x :: append\ xs\ ys
\end{array}
$$

The return type contains an addition expression $n + m$ which needs evaluation to get its value. For example, if we append $\langle 1, 2\rangle$ of type $Vec\ 2$ with $\langle 3\rangle$ of type $Vec\ 1$, the resulting list $\langle 1, 2, 3\rangle$ will have type $Vec\ (2 + 1)$. We need to further perform the computation $2 + 1 = 3$ to eliminate the addition form and obtain the final type $Vec\ 3$. In contrast, traditional languages rarely involve computations at the type level and usually have a strong distinction between terms and types.

Benefits of Dependent Types. Dependent types have been drawing a lot of attention in 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 and Weirich 2015; Sjöberg 2015; Eisenberg 2016; Casinghino 2014; Weirich et al. 2017]. A key reason why dependent types are interesting is that they naturally lead to a unification between types and terms, which enables both additional *expressiveness* and *economy of concepts*. Dependent type systems are more expressive because types can carry more information about terms. For example, by using vector types Vec, we can statically know the length of lists. In contrast, if using conventional list types, such as `List<T>` in Java, we can only know their lengths at *run-time* through the `size()` method. Thus, the dependent list type provides more refined type information (the exact list length) and is helpful to prevent more errors earlier at *compile-time*, such as out-of-bounds errors.

The other potential benefit of dependent types comes from the fact that as terms are allowed to occur in types, the strong separation between terms and types is no longer necessary. Once various different levels of syntax (such as terms and types) are unified, the redundancy of language constructs at the various levels can be avoided. For example, Java uses a special syntax for instantiating generics, e.g., `List<Object>`, which is different from an ordinary method call such as `Arrays.sort(a)`. The type operator `List` is essentially a *type-level function* and its instantiation can be seen as a function application. Supposing that we unify term and type levels in Java, we can instantiate generics just like calling methods, e.g., `List(Object)`. This leads to an *economy of concepts*: with unified syntax, there can be just one form of functions for both type-level functions (generics) and term-level functions (methods). Moreover, traditional functional languages like Haskell [Marlow et al. 2010] and ML [Milner et al. 1990] that support higher-kinded types have even more levels in the stratified syntax, i.e., terms, types and kinds. Unifying syntactic levels in such languages can result in a significantly more compact metatheory, and can also lead to a reduction of necessary implementation effort.

Benefits of Combining Type Features. Modern programming languages tend to support multiple type features simultaneously. Combining type features enables the possibility of writing programs in different programming paradigms. For example, Scala supports both subtyping and higher-kinded types. Not only can we write Scala programs in the object-oriented style, but also in the functional style, e.g., defining a `Monad` trait in Scala as follows:

```
trait Monad[M[_]] {
  def ret[A](a: A): M[A]
  def bind[A, B](m: M[A], f: A => M[B]): M[B]
}
```

The `Monad` trait follows the $Monad$ typeclass definition in Haskell and can also quantify over type constructors, such as the `Option[T]` type in Scala.

Moreover, with different type features combined, the language becomes more expressive and enables more programming features. For example, combing subtyping and higher kinds enables *higher-order subtyping* in Scala, which allows subtyping relations between type constructors. We can define a non-empty list type `Cons[T]`, which is a subtype of the general linked list type `L[T]`:

```
trait L[T] {
    def isEmpty: Boolean
    def head: T
    def tail: L[T]
}

class Cons[T](val head: T, val tail: L[T]) extends L[T] {
    def isEmpty = false
}
```

1.2 The Design Space of Dependently Typed Languages

Conventionally, dependent types are used as a logic to implement programming languages aiming at *theorem proving*, e.g., Twelf [Pfenning and Schürmann 1999] and Coq [The Coq development team 2016]. This is made possible by the *Curry-Howard correspondence* [Howard 1980] which connects the world of logics and the world of programming languages. The correspondence states that propositions in logic can be viewed as types and proofs can be viewed as program values. The expressiveness of dependent types makes languages like Twelf and Coq suitable as *proof assistants*. With dependent types, types can mention terms and correspondingly we can illustrate useful propositions containing terms. For example in Coq, we can write equality propositions to describe properties of additions such as $a + b = b + a$ for commutativity and $(a + b) + c = a + (b + c)$ for associativity.

To serve as proof assistants, languages like Twelf and Coq are also required to be *consistent* when viewed as logics. Logical consistency is critical to proof assistants, which guarantees that proofs will not go wrong, e.g., one can never derive a term of type $False$ or prove an absurd result $True = False$. Logical consistency is often ensured by *strong normalization*, a property that evaluating well-formed terms or types always terminates.

Apart from the common logical use, dependent types can also be beneficial to the area of traditional general-purpose programming. Dependently typed languages like Coq, Agda and Idris are designed to be both proof assistants and programming languages. However, the combination of dependent types and traditional programming features may cause problems. Traditional languages allow users to write *recursive programs* without restrictions. This is not the case in languages like Coq, Agda and Idris.

Otherwise, non-terminating recursive programs would break crucial properties of proofs assistants, such as strong normalization, logical consistency or *decidable type checking*, i.e. the type

checking algorithm always terminates. At the type level, non-terminating recursive functions can show up in definitions of dependent types. Strong normalization no longer holds since evaluation of such types may not terminate. Type checking may also involve the evaluation of non-terminating types and becomes undecidable. At the term level, non-terminating programs can be used to prove arbitrary theorems. For example, we can build a fixpoint combinator fix using a recursive definition:

$$\mathit{fix} : (A \to A) \to A$$
$$\mathit{fix}\ f = f\ (\mathit{fix}\ f)$$

With fix, we can prove an absurd theorem $\mathit{True} = \mathit{False}$, i.e., find a term with such type:

$$\mathit{fix}\ \mathit{id} : (\mathit{True} = \mathit{False})$$
$$\textbf{where}\ \mathit{id} = \lambda x : (\mathit{True} = \mathit{False}).\ x$$

This leads to logical inconsistency since we can prove a wrong result $\mathit{True} = \mathit{False}$.

Termination and Positivity Checking. Generally determining whether a recursive function terminates or not is essentially the *halting problem*, which is *undecidable* [Turing 1937]. Dependently typed languages usually take a conservative approach by limiting how recursive programs and types can be written, enforced by various checks including *termination checking* and *positivity checking*.

Termination checking requires recursive functions to have specific forms that are known to terminate. For example, Coq employs a restriction called *primitive recursion* [Bertot and Castran 2010; Chlipala 2013], which only allows recursive calls on syntactic sub-terms of the original arguments. Terminating total functions that are not primitive recursive cannot be directly written in Coq, e.g., the Ackermann function [Ackermann 1928; Bertot and Castran 2010]:

$$\begin{array}{lcl} \mathit{Ack}(0, n) & = & n + 1 \\ \mathit{Ack}(m + 1, 0) & = & \mathit{Ack}(m, 1) \\ \mathit{Ack}(m + 1, n + 1) & = & \mathit{Ack}(m, \mathit{Ack}(m + 1, n)) \end{array}$$

Positivity checking forbids recursive datatypes that occur in negative positions since such datatypes allow users to write non-terminating functions. The following datatype definition is a simple example that violates the positivity checking of Coq [Bertot and Castran 2010]:

```
Inductive T : Set := MkT : (T -> T) -> T.
```

but can be defined in Haskell without any issue:

$$\textbf{data}\ T = \mathit{MkT}\ (T \to T)$$

Issues of Termination and Positivity Checking. The termination and positivity checkers make these dependently typed languages non-starters for traditional programming styles based on general recursion. Programmers need to struggle with these checkers to let programs compile. Termination checkers are usually not intelligent enough to detect terminating programs, such as the Ack function. Another example is the implementation of merge sort in Coq. The following definition intuitively implements the algorithm of merge sort, which is terminating, but cannot pass the termination checker [Chlipala 2013]:

```
Fixpoint mergeSort (xs : list A) : list A :=
    if leb (length xs) 1
      then xs
      else let xss := split xs in
        merge (mergeSort (fst xss)) (mergeSort (snd xss)).
```

In contrast, Haskell programmers can write a recursive merge sort function more freely without any restriction:

$$
\begin{aligned}
&mergeSort\ [\,] &&= [\,]\\
&mergeSort\ [x] &&= [x]\\
&mergeSort\ xs &&= \mathbf{let}\ (a, b) = split\ xs\\
&&&\quad \mathbf{in}\ \ merge\ (mergeSort\ a)\ (mergeSort\ b)
\end{aligned}
$$

Our Focus. The design space between traditional languages and full-spectrum dependently typed languages is very large. It would be interesting if there are language designs that employ dependent types and benefit from the added expressiveness and economy of concepts, but also enjoy the liberty of writing recursive functions and types without termination or positivity checking. Previously, researchers have explored such languages designs and proposed dependently typed calculi with general recursion, such as Cayenne [Augustsson 1998] and Zombie [Casinghino et al. 2014; Sjöberg and Weirich 2015]. They showed the feasibility of combining dependent types with general recursion for general-purpose programming.

This thesis mainly focuses on such design space. We explore designs of languages that *employ dependent types, but aim at traditional programming instead of theorem proving*. In such languages, general recursive programs and types can be freely written. Decidable type checking is a desirable property, while strong normalization or logical consistency are not guaranteed. The termination/positivity checking can be avoided, which makes these languages less restrictive in terms of writing recursive programs and thus more friendly to end users. However, dependent types do not come for free and several challenges may arise when combining dependent types with other programming features, as discussed later in the next section.

1.3 Motivations and Challenges

Our goal of this thesis is to explore how to combine dependent types with traditional programming features in *simple yet expressive* calculi that are friendly to formalization, implementation and extension. In this section, we discuss the motivations and challenges of combining dependent types with various features, including *general recursion*, *subtyping* and *strong dependent sums*.

1.3.1 Unified Syntax, General Recursion and Decidable Type Checking

Dependent types enable unifying types and terms. By employing unified syntax with dependent types, it is possible to design a calculus that is both simple and expressive. One existing example is the *Pure Type Systems* (PTS) [Barendregt 1991, 1992] that have only one syntactic level for both terms and types. PTS has a very concise presentation consisting of only 5 language constructs and 7 typing rules. It is also powerful to subsume a wide range of typed lambda calculi, including the calculus of constructions [Coquand and Huet 1988] containing dependent types.

Though PTS does not contain any explicit forms of recursion and is typically used as a framework to study logic systems, the approach of PTS-style unified syntax can be helpful

for traditional language designs. Unifying different syntactic levels removes the duplication of constructs and reduces the effort of formalization and implementation. Not surprisingly, researchers have in the past proposed several dependently typed languages with unified syntax and general recursion. These languages can both model the traditional programs with recursion and allow using dependent types to write expressive and type-safe programs that cannot usually be done in traditional languages. For example, Cayenne [Augustsson 1998] is a Haskell-like dependently typed language with unified syntax. Cayenne allows users to write a *type-safe* format printing function, i.e., `printf`, which has the following type:

```
printf :: (fmt :: String) -> PrintfType fmt
```

The type of `printf` is a dependent type whose return type is determined by the `fmt` string. A type-level function `PrintfType` *recursively* computes the type of `printf` according to the provided `fmt`. Thanks to the unified syntax, it is defined just like any other recursive functions:

```
PrintfType :: String -> *
PrintfType ""           = String
PrintfType ('%':'d':cs) = Int ->    PrintfType cs
PrintfType ('%':'s':cs) = String -> PrintfType cs
PrintfType ('%': _ :cs) =           PrintfType cs
PrintfType ( _ : cs   ) =           PrintfType cs
```

`PrintfType` ensures that `printf` only accepts arguments with correct types specified by the format string. For example, when `fmt` is `"%d"`, we have

```
printf "%d" :: Int -> String
```

which only accepts an integer and prevents malformed inputs. Note that the type-safe `printf` example only requires a few, *finite* steps of type-level computation. It can actually be simulated in Haskell with other existing type-level features such as typeclasses or Template Haskell [Sheard and Jones 2002]. Nonetheless, the simulation is not as convenient or intuitive as programming in Cayenne directly using dependent types.

Challenges. However, in dependently typed calculi with unified syntax, it is non-trivial to simultaneously integrate general (unrestricted) recursion and retain *decidable type checking*. For example, type checking of Cayenne is undecidable. The reason is that as dependent types involve type-level computation, the type checker may get stuck if trying to evaluate non-terminating terms that show up in types. For example, recall the fixpoint combinator in Section 1.2. If there exists a vector type with fix as follows:

$$\mathit{Vec}\ (\mathit{fix}\ (\lambda x : \mathit{Nat}.\ x + 1))$$

The type checker will get stuck when evaluating the parameter of Vec, which is a non-terminating operation that infinitely adds one to the result. Moreover, unified syntax removes the syntactic boundary between terms and types, allowing us to use the same fix combinator to directly construct a recursive type:

$$\mathit{fix}\ \mathit{id} : \star$$
$$\textbf{where}\ \mathit{id} = \lambda x : \star.\ x$$

The type $\mathit{fix}\ \mathit{id}$ can be unfolded indefinitely, which may also block the type checker. For languages with stratified syntax, we can still make syntactic restrictions to prevent recursive terms (e.g. fix)

being directly used as types. But this is usually difficult for languages with unified syntax whose terms can be freely lifted to the type level as types.

Besides Cayenne, there are several other dependently typed calculi supporting unified syntax and general recursion, such as the calculus by Cardelli [1986b] and the $\Pi\Sigma$ calculus by Altenkirch et al. [2010]. But both calculi fail to retain decidable type checking similarly to Cayenne. For pragmatic reasons, some theorem provers with unified syntax, e.g., Idris and Agda, provide an option to turn off the termination checker and allow general recursion, but the type checking becomes undecidable. More recently, several studies have proposed dependently typed calculi with unified syntax and also managed to preserve decidable type checking in the presence of general recursion. Notable examples are the Zombie language from the Trellys Project [Sjöberg et al. 2012; Casinghino et al. 2014; Sjöberg and Weirich 2015; Casinghino 2014; Sjöberg 2015] and System DC from Dependent Haskell [Weirich et al. 2017]. However, these studies have more ambitious goals and require more sophisticated mechanisms, such as logical subsystems and type equality. The resulting calculi are significantly more complex when compared to systems based on PTS.

1.3.2 Combining Dependent Types and Subtyping

Besides functional programming in Haskell and ML, object-oriented programming (OOP) is a popular programming paradigm with "objects" at the core to encapsulate both data and programs. Beyond the concept of objects, type systems for modern OOP languages are becoming more and more complex. For example, Java was initially designed to be a relatively simple OOP language that tried to address many issues of C++ caused by its complexity. The core features of (original) Java can be formalized in a simple calculus called *Featherweight Java* (FJ) [Igarashi et al. 1999]. The syntax of classes in FJ is simple:

$$L ::= \mathbf{class}\ C\ \mathbf{extends}\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}$$

With the growth of the language, many new features were introduced into modern Java. One notable feature is *parametric polymorphism*, also known as generics, which is introduced in Java 5. To model generics, the original FJ is extended to *Featherweight GJ* (FGJ) [Bracha et al. 1998]. The syntax of classes in FGJ becomes more complex ($\triangleleft$ is the abbreviation of $\mathbf{extends}$):

$$L ::= \mathbf{class}\ C\langle\overline{X} \triangleleft \overline{N}\rangle \triangleleft N\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}$$

Moreover, Java continues evolving with more features, such as lambda expressions in Java 8 and local type inference in Java 10 [Goetz 2016], thus requires further extensions of core calculi to formalize these features. Another example is Scala, which supports even more advanced features such as *higher-order polymorphism* [Girard 1972; Moors et al. 2008], *type members* [Odersky et al. 2003] and *path-dependent types* [Odersky et al. 2004; Rompf and Amin 2016]. The extra complexity of the type systems is reflected by the significant effort to develop the corresponding formal theory. A notorious example of this is the development of the foundational metatheory for Scala, which has been an ongoing effort that lasted for more than 10 years and recently culminated with the Dependent Object Types (DOT) calculus [Amin et al. 2012a].

Like functional languages, OOP languages can also benefit from dependent types for both added expressiveness and economy of concepts. Given that the complexity of type systems for

OOP languages is so high, techniques for bringing down such complexity, while retaining or even increasing expressiveness are certainly welcome. The economy of concepts afforded by unified syntax typical of dependently typed languages can help here, since it can significantly reduce the number of language constructs and relations needed in a calculus. Unfortunately, there has been less work on dependently typed calculi for OOP. There are mainly two challenges when developing such calculi. One challenge we already mentioned is the interaction between general recursion and dependent types. This applies to programming languages in general, and is not limited to OOP languages.

The other challenge arises from the combination of subtyping and dependent types. OOP supports polymorphism based on the subtyping relation. However, subtyping is also a substantial difference to traditional calculi based on PTS which typically do not support such feature. The issue with subtyping is well summarized by Aspinall and Compagnoni [1996]:

> *One thing that makes the study of these systems difficult is that* **with dependent types, the typing and subtyping relations become intimately tangled**, *which means that tested techniques of examining subtyping in isolation no longer apply.*

In essence the big difficulty is that the introduction of dependent types makes typing and subtyping depend on each other. This causes several difficulties in developing the metatheory for calculi that combine dependent types and subtyping. The metatheory becomes quite complex in some calculi [Aspinall and Compagnoni 1996; Castagna and Chen 2001], e.g., the transitivity of subtyping is entangled with strong normalization. Most previous attempts [Aspinall and Compagnoni 1996; Zwanenburg 1999; Castagna and Chen 2001; Chen 1997, 2003] try to address the problem by *untangling* typing and subtyping, so that the metatheory of subtyping can be developed before the metatheory of typing. One interesting exception is the Pure Subtype Systems (PSS) by Hutchins [2010] which completely eliminate types and only contain the subtyping relation. An unfortunate result of these attempts is that several desirable language features can no longer be supported. For example, several systems [Aspinall and Compagnoni 1996; Zwanenburg 1999] drop the support of *top types*, which are essential in OOP programs to model the universal base class, e.g. the `Object` type in Java. PSS does not support contravariance when subtyping functions, and its metatheory is complex and not fully developed yet. No previous calculi have managed to subsume System $F_{\leqslant}$ [Cardelli et al. 1994] which is a canonical calculus capturing the essential OOP features.

1.3.3 Combining Dependent Types and Strong Sums

OOP languages like Scala and Rust [The Rust Project Developers 2011] support traits [Schärli et al. 2003] which are types that contain a set of methods. Scala and Rust also allow *type members* (also called associated types) in traits, which are abstract types and useful to define generic datatypes in a more *modular* way. For example, we can define an abstract trait of integer sets using type members in Scala:

```
trait Set {
    type T
    def empty(): T
    def member(x: Int, s: T): Boolean
    def insert(x: Int, s: T): T
}
```

The type member `T` represents the abstract type of a set implementation and can be mentioned by methods. Alternatively, traits with type members can be formulated as traits with type parameters (i.e. generics):

```
trait Set[T] {
    def empty(): T
    def member(x: Int, s: T): Boolean
    def insert(x: Int, s: T): T
}
```

However, abstract types (i.e. `T`) need to be *exposed* out of traits with generics. When referring to `Set[T]`, one needs to provide an actual `T` in advance to obtain a complete type. In contrast, if formulated as type members, abstract types can be *hidden* inside traits. Thus, traits carrying with type members can have higher cohesion thus better *modularity*.

To access abstract type members, one needs certain forms of dependent types. In Scala, this is done by a weaker form of dependency, called *path dependency* [Amin et al. 2012a], which only allows types to depend on "paths", i.e., a chain of member access. For example, consider a generic function for testing if an element is in an abstract set:

```
def isMember (s: Set)(x: Int, t: s.T) = s.member(x, t)
```

The last parameter has the type `s.T`, which depends on a term, the first parameter `s`. The type `s.T` is called a *path-dependent type* [Amin et al. 2014]. DOT utilizes path dependent types to model the access of abstract type members. However, DOT contains many features, such as a rich notion of type bounds, aiming at modeling more features of Scala than only type members. It has a complex metatheory that requires several new proof techniques [Rompf and Amin 2016; Rapoport et al. 2017].

If we only focus on encoding type members, we can alternatively use *strong dependent sums* (or simply strong sums) in traditional dependently typed calculi. Dependent sums are generalized pairs whose second component can depend on the first. For example, we can encode the `Set` trait using a dependent sum type (i.e. Σ-type) with a record type:

$$
\begin{array}{l}
Set = \Sigma\, T : \star.\ \{ \\
\quad empty : \quad T, \\
\quad member : Int \to T \to Bool, \\
\quad insert : \quad Int \to T \to T \\
\}
\end{array}
$$

Strong sums allow us to use *projections* to access the components. Given a term $s : Set$, the second projection $s.2$ extracts the record and $(s.2).member$ further extracts the function from the record. Moreover, strong sums can be used to model some other modular structures, such as module systems of ML languages [MacQueen 1986; Milner et al. 1990].

However, combining strong sums in dependently typed calculi is also non-trivial. In traditional dependently typed calculi that are *impredicative*, such as the calculus of constructions [Coquand and Huet 1988], the introduction of strong sums leads to logical inconsistency [Coquand 1986; Hook and Howe 1986] and hence the loss of strong normalization. This makes it hard to preserve desirable properties, especially in dependently typed calculi with subtyping (e.g. $\lambda P_{\leqslant}$ by Aspinall and Compagnoni [1996]), whose properties such as transitivity and subject reduction rely on strong normalization. Previous studies try to address this issue by dropping [Harper and Mitchell 1993]

or partially dropping [Stump 2017; Bowman et al. 2017] the impredicativity, but the expressiveness of these calculi is also reduced.

1.4 Our Proposals

To address the challenges posed by combining dependent types with traditional programming features, we propose three novel calculi, namely *Pure Iso-Type Systems*, *the* $\lambda I_{\leqslant}$ *calculus* and *the* λI_{Σ} *calculus*. In these calculi, we manage to combine dependent types with features such as general recursion, subtyping and strong sums by proposing several novel techniques, including *iso-types*, *unified subtyping* and *iso-strong sums*. We mainly focus on traditional programming scenarios that require *few* type-level computations, which fit well with our treatment of dependent types. The proposed calculi enjoy both *expressiveness* and *simplicity*. By employing the unified syntax of dependent types, the calculi are more concise and expressive than traditional non-dependent calculi. Several advanced features of traditional languages, such as higher-kinded polymorphism and higher-order subtyping, can be supported with minimal effort. The calculi are also simple enough to enable the full development of formal metatheory and the possibility of extensions. In the rest of this section, we briefly introduce our proposed calculi and techniques.

1.4.1 Pure Iso-Type Systems and Iso-Types

Pure Iso-Type Systems (PITS) are a family of calculi that employ unified syntax, support general recursion and preserve decidable type checking. PITS has a comparable level of simplicity to PTS, consisting of only 8 language constructs and no more than 9 typing rules. The key to retaining decidable type checking in the presence of general recursion is *iso-types*, a generalization of iso-recursive types [Crary et al. 1999; Pierce 2002]. Iso-types view not only folding and unfolding of recursive types but also any types that are (beta) convertible as isomorphic. Unlike PTS, there is no implicit type-level computation in PITS. Each type-level computation step is *explicitly* triggered by a term-level type-safe cast operator. Single-step computation is trivially terminating — it stops after one step, thus type checking is decidable even with non-terminating programs at the type level. Meanwhile, term-level programs using general recursion can be non-terminating as in traditional languages. There is no termination checking or positivity checking in PITS.

We emphasize that PITS does sacrifice some convenience when performing type-level computations in order to gain the ability of doing arbitrary general recursion at the term level. The goal of proposing PITS is to show the use of PTS-style unified syntax for benefiting language designs intended for *traditional programming* that only involves *lightweight* type-level computations, but not intended for full-spectrum dependently typed programming that requires intensive type-level computations. The design of PITS is suitable to serve as a foundation for languages in-between traditional ones (e.g. ML and Haskell) and full-spectrum dependently typed ones (e.g. Idris and Agda). To illustrate the feasibility of using PITS as such a foundation, we build a simple surface language called **Fun**. We show several advanced language features such as higher-kinded types can be encoded in PITS by examples written in **Fun**.

PITS also enjoys the flexibility of choosing different reduction relations for the explicit cast operators. We study three variants of PITS, namely the *call-by-name* PITS, the *call-by-value* PITS and the *full* PITS, which differ on the reduction strategy used by casts. The call-by-name/value PITS uses conventional call-by-name/value reduction rules for casts. This leads to a relatively

simple metatheory that has direct proofs of properties. In contrast, the full PITS uses the more expressive parallel reduction for casts, which can perform full beta-reduction. But the cost is the complex metatheory of full PITS that requires indirect proofs from an auxiliary system erasing all casts. For all variants, we prove important results of metatheory, including type-safety and decidability of type checking.

1.4.2 The $\lambda I_{\leqslant}$ Calculus and Unified Subtyping

The $\lambda I_{\leqslant}$ calculus is a dependently typed calculus with subtyping. To address the issues arising from the combination of dependent types and subtyping, $\lambda I_{\leqslant}$ employs a novel technique called *unified subtyping*, which unifies both typing and subtyping relations. The unified subtyping relation can be viewed as a typed version of the conventional subtyping relation.

In $\lambda I_{\leqslant}$, there is only one single form of judgment, i.e., the unified subtyping judgment. There is no separated typing or type well-formedness judgment, both of which are syntactic sugar of the unified subtyping judgment. The issue of mutual dependency between typing and subtyping, which is a major problem in other formalizations [Aspinall and Compagnoni 1996; Castagna and Chen 2001; Zwanenburg 1999] of calculi with dependent types and subtyping, does not exists in $\lambda I_{\leqslant}$, because the typing relation is essentially the (unified) subtyping relation itself. In contrast to previous work that struggles with untangling typing and subtyping, we propose a different approach that embraces such entanglement. Our approach is also fundamentally different from Hutchins' PSS since $\lambda I_{\leqslant}$ still retains types.

Meanwhile, $\lambda I_{\leqslant}$ is a dependently typed calculus and can be viewed as a variant of PITS. It features the PTS-style unified syntax and employs the iso-type approach to address the issues of combining recursion and dependent types. Moreover, $\lambda I_{\leqslant}$ supports extra OOP features over PITS, including higher-order subtyping [Pierce and Steffen 1997], bounded quantification [Cardelli et al. 1994] and top types. It can fully subsume System $F_{\leqslant}$ and enjoys several standard and desirable properties, such as type-safety and transitivity of subtyping.

1.4.3 The λI_{Σ} Calculus and Iso-Strong Sums

The λI_{Σ} calculus is a dependently typed calculus with strong sums. It is a variant of $\lambda I_{\leqslant}$ and also based on the ideas of unified subtyping and iso-types. To address the challenges posed by combining dependent types and strong sums, we propose *iso-strong sums* whose strong destructors (e.g. the second projection) are typed using iso-types. In λI_{Σ}, cast operators employ the *call-by-value* reduction, similarly to the one of call-by-value PITS. The typing results of the strong sum destructors are intermediate *type-level applications* instead of standard direct substitutions. With such change, all type-level computation for strong sums can be performed by just call-by-value casts. There is no need of using full casts from the full PITS, which bring extra complexity in the metatheory.

At the same time, iso-types and iso-strong sums in $\lambda I_{\leqslant}$ decouple properties from strong normalization. This makes it possible to support impredicative polymorphism in the presence of iso-strong sums and retain desirable properties. λI_{Σ} enjoys the same desirable properties as $\lambda I_{\leqslant}$, including type-safety and transitivity, both of which can be proved without requiring strong normalization. For demonstration purposes, we also build a lightweight surface language **Sig** on top of λI_{Σ}. We show how Scala-like type members and traits can be encoded by strong sums of λI_{Σ} via a type-sound elaboration semantics of **Sig**.

1.5 Contributions and Outline

Contributions. In summary, the main contributions of this thesis are:

- **Pure Iso-Type Systems:** A variant of PTS with general recursion, unified syntax and decidable type-checking. We prove type-safety and decidability of type checking for all three variants of PITS.
- **Iso-Types:** A generalization of iso-recursive types, which makes all type-level computation steps explicit via cast operators. The combination of casts and recursion subsumes iso-recursive types.
- **Reduction Strategies of PITS:** We study PITS with three reduction strategies: call-by-name PITS, call-by-value PITS and PITS with parallel reduction at the type-level. We show the trade-offs between the simplicity and expressiveness for different reduction strategies.
- **Unified subtyping:** A novel technique that unifies typing and subtyping into a single relation. This technique enables the development of expressive dependently typed calculi with subtyping.
- **The $\lambda I_{\leqslant}$ calculus:** A dependently typed calculus with subtyping that employs unified syntax, iso-types and unified subtyping. The calculus supports top types, higher-order polymorphism and bounded quantification and can fully subsume System $F_{\leqslant}$. We prove transitivity of subtyping and type-safety for this calculus.
- **Iso-Strong Sums:** A novel treatment of dependent sums whose strong destructors are typed as intermediate type-level applications. This makes it possible to use call-by-value casts for required type-level computations.
- **The λI_{Σ} calculus:** A dependently typed calculus with unified subtyping and iso-strong sums. This calculus supports impredicative polymorphism and retains desirable properties. We prove transitivity of subtyping and type-safety for this calculus.

Outline of Thesis. The thesis is divided into two parts. The first part presents an informal overview of iso-types by motivating examples (Chapter 3), and the formal theory of Pure Iso-Type Systems (Chapter 4). It is the foundation of the thesis and provides a substantial theory for iso-types. The second part further studies calculi with both iso-types and subtyping. It presents two new calculi, i.e., the $\lambda I_{\leqslant}$ calculus with unified subtyping (Chapter 5) and the λI_{Σ} calculus with iso-strong sums (Chapter 6). In summary, the remaining chapters of this thesis are organized as follows:

Chapter 2 provides several background knowledge for this thesis.

Chapter 3 gives an overview of iso-types. We discuss the motivation for the development of iso-types and informally introduce the basic mechanism of iso-types in PITS. We give examples written in the surface language **Fun** to illustrate the usefulness and expressiveness of PITS.

Chapter 4 gives a formal presentation of Pure Iso-Type Systems (PITS), including three variants of PITS that differ on the reduction strategy used in explicit casts. We present the formal

syntax, operational semantics and typing judgments. We develop metatheory for all variants and prove both type-safety and decidability of type checking. We provide an extensive comparison of PITS to a closely related calculus called PTS$_f$ [van Doorn et al. 2013].

Chapter 5 presents the $\lambda I_{\leqslant}$ calculus. We discuss the motivation of unified subtyping and present an example of object encodings using $\lambda I_{\leqslant}$. We formally present the specification of $\lambda I_{\leqslant}$. Two major results of the metatheory, namely transitivity of subtyping and type-safety are proved. We present a bidirectional version of $\lambda I_{\leqslant}$ and show the subsumption of System $F_{\leqslant}$ in $\lambda I_{\leqslant}$. We discuss several alternative designs of $\lambda I_{\leqslant}$.

Chapter 6 presents the λI_{Σ} calculus. We discuss the motivation of iso-strong sums and present an example of encoding Scala-like traits using λI_{Σ}. We present the formal specification of λI_{Σ}. We discuss its metatheory and prove major results including transitivity and type-safety. We also show **Sig**, a lightweight surface language over λI_{Σ} with Scala-like traits for presenting the application of strong dependent sums. We show a type-directed translation from **Fun** to λI_{Σ} and prove its soundness.

Chapter 7 discusses the related work to PITS, $\lambda I_{\leqslant}$ and λI_{Σ}.

Chapter 8 concludes the thesis and discusses potential directions for future work.

Mechanized Proofs and Prototype Implementation. All proofs in the thesis are machine-checked in Coq theorem prover [The Coq development team 2016], except for the completeness of λI_{Σ} over System $F_{\leqslant}$ and the translation soundness of **Sig** which can be found in Appendix A. These proofs are manual due to the difficulties in mechanizing proofs between different type systems [Kaiser et al. 2017]. For lemmas that are proved manually, we add a pencil symbol "✎" in the title (e.g. Lemma 5.5.1). We also implement a prototype interpreter and compiler of **Fun**, a simple language which features algebraic datatypes and pattern matching by elaborating to PITS. The mechanized proofs and the **Fun** implementation are available online at https://bitbucket.org/ypyang/archive.

Prior Publications. The content of this thesis is partially based on previously published papers as follows:

- **Yanpeng Yang**, Xuan Bi, and Bruno C. d. S. Oliveira. 2016. Unified Syntax with Iso-types. In *Programming Languages and Systems*, Atsushi Igarashi (Ed.). Springer International Publishing, Cham, 251-270.

This paper is the basis of Chapter 3 and Chapter 4. The material of the paper is significantly revised and expanded in the thesis. The calculus from the paper is further generalized to a family of calculi with the PTS tradition called PITS. The study of call-by-value PITS and the completeness of full PITS to PTS are new in Chapter 4.

- **Yanpeng Yang** and Bruno C. d. S. Oliveira. 2017. Unifying Typing and Subtyping. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 47 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133871

This paper is the basis of Chapter 5. The study of the combination of unified subtyping and strong sums is new in Chapter 6.

CHAPTER 2

BACKGROUND

In this chapter, we introduce several preliminaries on type theory which are basic concepts for better understanding the thesis, including pure type systems (Section 2.1), dependent sums (Section 2.2), iso-recursive types (Section 2.3), subtyping (Section 2.4) and path-dependent types (Section 2.5). Note that all topics introduced in this chapter are well-established and well-studied concepts of type theory. Readers who are familiar with these topics can quickly go through this chapter. For readers who may not be familiar with notations of type theories, we recommend the book *Types and Programming Languages* by Benjamin C. Pierce [2002] for a quick introduction.

2.1 Pure Type Systems

Pure Type Systems (PTS) [Barendregt 1991, 1992] are a generic framework to study a family of type systems, including the simply typed lambda calculus ($\lambda\!\rightarrow$), System F [Reynolds 1974; Girard 1972] ($\lambda 2$) and the calculus of constructions (λC) [Coquand and Huet 1988]. PTS unifies terms and types (also kinds) into a single syntactic category. Thanks to the unified syntax, PTS has a very concise presentation, which consists of only seven typing rules. Two of them are parameterized by the specification of PTS, i.e., the axioms for typing sorts, and the rules for typing Pi-types. By instantiating the specification, we can obtain a wide range of type systems, even logically inconsistent ones. The content of this section is heavily inspired by previous presentations of PTS by Barendregt [1992] and Severi and de Vries [2012].

2.1.1 Basics of PTS

Specification. The specification of PTS is defined as follows [Severi and de Vries 2012]:

Definition 2.1.1 (Specification of PTS)**.** *PTS is specified by a triple* $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ *where*

1. $\mathcal{S}$ *is a set of constants called* sorts*;*
2. $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ *is a set of* axioms*;*
3. $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ *is a set of* rules*.*

We use the metavariable $s \in \mathcal{S}$ to range over sorts. There is a special kind of PTS specification, called the *functional* or *singly-sorted* specification:

Definition 2.1.2 (Functional Specification)**.** *A PTS specification is* functional *if*

1. *If* (s_1, s_2) *and* (s_1, s_2') *are in* $\mathcal{A}$*, then* $s_2 = s_2'$.
2. *If* (s_1, s_2, s_3) *and* (s_1, s_2, s_3') *are in* $\mathcal{R}$*, then* $s_3 = s_3'$.

The functional specification is useful to prove properties such as typing uniqueness and decidability of type checking for PITS, as discussed in Section 4.1.5.

Syntax. In PTS, terms and types are defined in the same syntactic category:

Definition 2.1.3 (Syntax of PTS)**.** *The syntax of PTS is defined by*

$$\begin{array}{lll} a, b, A, B & ::= & s \mid x \mid a\ b \mid \lambda x : A.\ a \mid \Pi x : A.\ B \\ \Gamma & ::= & \varnothing \mid \Gamma, x : A \end{array}$$

where a, b, A, B *denote the pseudo-terms and* Γ *denotes the pseudo-contexts.*

The variables are denoted by x, y, z, etc. Although there is no distinction between terms and types, by convention we still use lower-case meta-variables a, b, c, etc. to denote terms and upper-case meta-variables A, B, C, etc. to denote types. We use $\mathsf{FV}(a)$ for the set of free variables in a. We use syntactic sugar $A \to B$ to denote $\Pi x : A.\ B$ when $x \notin \mathsf{FV}(B)$, i.e., x does not occur free in B. Given $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$, the domain of the context is defined by $\mathsf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$.

Dynamic Semantics. The beta-reduction of PTS is defined as follows [Severi and de Vries 2012]:

Definition 2.1.4 (One-step Beta-reduction)**.** *Given the beta rule:*

$$(\lambda x : A.\ b)\ a \rightsquigarrow_\beta b[x \mapsto a]$$

The one-step beta-reduction $\longrightarrow_\beta$ *is defined as the smallest relations on pseudo-terms that are closed under the beta rule (*$\rightsquigarrow_\beta$*) and under contexts.*

where the notation $b[x \mapsto a]$ denotes a capture-avoiding substitution that replaces all free occurrences of x in b with a. The relation $\longrightarrow_\beta$ is inductively defined by applying the base case, i.e., the beta rule ($\rightsquigarrow_\beta$), for every sub-term of a pseudo-term. Thus, we also call $\longrightarrow_\beta$ *full* beta-reduction, since the beta rule can be applied at any position of a (pseudo) term. Based on $\longrightarrow_\beta$, we define the multi-step beta-reduction and beta-equivalence:

Definition 2.1.5 (Multi-step Beta-reduction)**.** *The multi-step beta-reduction* $\twoheadrightarrow_\beta$ *is the reflexive-transitive closure of* $\longrightarrow_\beta$.

Definition 2.1.6 (Beta-equivalence)**.** *The beta-equivalence* $=_\beta$ *is the reflexive-symmetric-transitive closure of* $\longrightarrow_\beta$*, i.e., the smallest equivalence relation containing* $\longrightarrow_\beta$.

Typing Rules. A PTS determined by the specification $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ has the notation λS and is defined by the judgment $\Gamma \vdash_S a : A$, or simply $\Gamma \vdash a : A$. The notion of the judgment is defined by typing rules shown in Figure 2.1.

The axiom rule checks the type of sorts, which states that the type of sort s_1 is sort s_2 if the relation (s_1, s_2) is in the axiom set $\mathcal{A}$. The start rule types the variable x according to the binding

$$\text{(axiom)} \quad \varnothing \vdash s_1 : s_2 \qquad \text{if } (s_1, s_2) \in \mathcal{A}$$

$$\text{(start)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \qquad \text{if } x \text{ fresh in } \Gamma$$

$$\text{(weakening)} \quad \frac{\Gamma \vdash A : s \qquad \Gamma \vdash b : B}{\Gamma, x : A \vdash b : B} \qquad \text{if } x \text{ fresh in } \Gamma$$

$$\text{(product)} \quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.\ B) : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$\text{(abstraction)} \quad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.\ B) : s}{\Gamma \vdash (\lambda x : A.\ b) : (\Pi x : A.\ B)}$$

$$\text{(application)} \quad \frac{\Gamma \vdash b : (\Pi x : A.\ B) \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a : B[x \mapsto a]}$$

$$\text{(conversion)} \quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash a : B}$$

Figure 2.1. Typing rules of PTS

$x : A$ from the context. The weakening rule states that a typing judgment $\Gamma \vdash b : B$ still holds if the context is expanded with a well-formed binding $x : A$. Both start and weakening rules require that the new variable x adding to the context Γ should be fresh, i.e., $x \notin \mathsf{dom}(\Gamma)$. The product rule checks Pi-types and allows the binder type A, the body type B and the Pi-type itself to have different sorts. The relation among these sorts, i.e., (s_1, s_2, s_3), is defined by the rule set $\mathcal{R}$. The abstraction and application rules type check lambda abstractions and function applications, respectively. Finally, the conversion rule allows converting types that are beta-equivalent.

2.1.2 Examples of PTS

We show several concrete examples of PTS by instantiating different specifications.

The Lambda Cube. The lambda cube [Barendregt 1992] contains eight type systems of typed lambda calculi, as shown in Figure 2.2. The systems can be obtained by PTS specifications where

$$\mathcal{S} = \{\star, \Box\} \quad \mathcal{A} = \{(\star, \Box)\}$$

By convention, we use the abbreviation (s_1, s_2) for $(s_1, s_2, s_2) \in \mathcal{R}$. Ranging s_1 and s_2 over $\{\star, \Box\}$, there are four possible values of (s_1, s_2), denoted by

$$\mathcal{R}' = \{(\star, \star), (\Box, \star), (\star, \Box), (\Box, \Box)\}$$

Recalling the PTS typing rules for abstractions and products, each value represents a dependency pattern of functions:

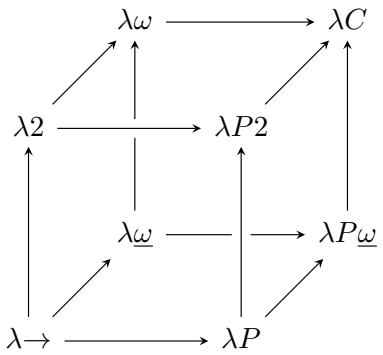

Figure 2.2. The lambda cube

Rule	Dependency	Example
$(\star,\star)$	terms depending on terms	ordinary functions
$(\Box,\star)$	terms depending on types	polymorphic functions
$(\star,\Box)$	types depending on terms	dependent types
$(\Box,\Box)$	types depending on types	type constructors

Note that there are eight subsets of $\mathcal{R}'$ containing the basic rule $(\star,\star)$, i.e., $(\star,\star) \in \mathcal{R} \subseteq \mathcal{R}'$, which give us the exactly eight calculi of the lambda cube. Among all eight calculi, we introduce four commonly used typed calculi:

- The *simply typed lambda calculus* ($\lambda\!\rightarrow$) is obtained by the PTS specification with the common $\mathcal{S}$ and $\mathcal{A}$ mentioned above, as well as $\mathcal{R} = \{(\star,\star)\}$ which is the smallest subset of rules.
- The *second-order lambda calculus* ($\lambda 2$), also known as System F [Reynolds 1974; Girard 1972], has the rule set $\mathcal{R} = \{(\star,\star),(\Box,\star)\}$, and additionally allows polymorphic functions over $\lambda\!\rightarrow$.
- The *higher-order polymorphic lambda calculus* ($\lambda\omega$), also known as System F_ω [Girard 1972], has the rule set $\mathcal{R} = \{(\star,\star),(\Box,\star),(\Box,\Box)\}$, and further supports type constructors over System F.
- The *calculus of constructions* (λC) [Coquand and Huet 1988] has the following rule set:

  $$\mathcal{R} = \{(\star,\star),(\Box,\star),(\Box,\Box),(\star,\Box)\}$$

  which is the largest subset of $\mathcal{R}'$, i.e., $\mathcal{R}'$ itself. λC further supports dependent types over System F_ω, which allow types to depend on terms.

Strongly Normalizing PTS. We have introduced the lambda cube that includes eight systems that are specialized PTSs. One common property is that all eight calculi are *strongly normalizing* [Barendregt 1992]. The definition is as follows:

Definition 2.1.7 (Normalization)**.**

1. *A term a is weakly normalizing if there exists a term b in normal form such that $a \twoheadrightarrow_\beta b$.*

2. *A term a is strongly normalizing if all beta-reduction sequences starting from a are finite.*

Definition 2.1.8 (Normalizing PTS). *A PTS λS is strongly (weakly) normalizing if for all $\Gamma \vdash a : A$ we have that a and A are strongly (weakly) normalizing.*

In general, if a type system is strongly normalizing, then it is consistent when viewed as a logic, i.e., it is *logically consistent*, meaning that there does not exist an absurd term of type $False = \Pi x : \star.\ x$. On the other side, we say that a system is *inconsistent* if all types are inhabited [Barendregt 1992; Girard 1972]. One can derive a term of $False$ type in an inconsistent system.

Inconsistent PTS. There are PTS specifications that can generate inconsistent systems. One well-known example is the $\lambda\star$ calculus [Barendregt 1992], which has the following PTS specification:

$$\mathcal{S} = \{\star\} \quad \mathcal{A} = \{(\star, \star)\} \quad \mathcal{R} = \{(\star, \star)\}$$

There is only one axiom called the "type-in-type" axiom [Cardelli 1986b] that causes circularity in typing sort, i.e. $\star : \star$. The inconsistency of $\lambda\star$ was first proved by Girard [1972]:

Theorem 2.1.1 (Girard's Paradox). *The type $False = \Pi x : \star.\ x$ is inhabited in $\lambda\star$, i.e. $\varnothing \vdash a : False$ for some a.*

The calculus is not normalizing, due to the following lemma [Barendregt 1992]:

Lemma 2.1.1 (Absurd Has No Normal Form). *Let λS be a PTS extending $\lambda 2$. Suppose $\varnothing \vdash a : False$. Then a has no normal form.*

By Girard's theorem, we can find a term a such that $\Gamma \vdash a : False$ in $\lambda\star$. Such a has no normal form and is not even weakly normalizing. Hence, the calculus is not weakly or strongly normalizing.

2.1.3 Metatheory of PTS

We state several important results of PTS. The properties hold for arbitrary PTS, unless otherwise stated. The following lemmas and theorems are from the work by Barendregt [1992] and proofs can be found in Barendregt's work or other related literature [van Benthem Jutting 1993; Barendregt 1991].

Lemma 2.1.2 (Substitution Lemma for PTS). *If $\Gamma, x : A, \Gamma' \vdash b : B$ and $\Gamma \vdash a : A$, then $\Gamma, \Gamma'[x \mapsto a] \vdash b[x \mapsto a] : B[x \mapsto a]$.*

Lemma 2.1.3 (Weakening Lemma for PTS). *Let Γ and Γ' be legal contexts such that $\Gamma \subseteq \Gamma'$. If $\Gamma \vdash a : A$, then $\Gamma' \vdash a : A$.*

Theorem 2.1.2 (Subject Reduction for PTS). *If $\Gamma \vdash a : A$ and $a \twoheadrightarrow_\beta a'$, then $\Gamma \vdash a' : A$.*

Theorem 2.1.3 (Decidability of Type Checking for Normalizing PTS). *If $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where $\mathcal{S}$ is finite, let λS be a PTS that is (weakly or strongly) normalizing. Then given Γ and a, it is decidable to check whether there exists an A such that $\Gamma \vdash a : A$ or not.*

Lemma 2.1.4 (Uniqueness of Types for Functional PTS). *Let λS be a PTS that is functional. If $\Gamma \vdash a : A$ and $\Gamma \vdash a : B$, then $A =_\beta B$.*

Notice that the decidability of type checking (Theorem 2.1.3) requires a PTS to be weakly or strongly normalizing. And the uniqueness of types (Lemma 2.1.4) requires a PTS to be functional (i.e. has a functional specification, see Definition 2.1.2).

2.2 Dependent Sums

A pair type, denoted by $A \times B$, is one of the simplest compound types. A pair of type $A \times B$ consists of two components where the first component has type A and the second has type B. In a dependent type theory, pair types can also be extended with type dependency, similarly to how dependent function types (i.e. Pi-types $\Pi x : A.\ B$) generalize non-dependent function types (i.e. arrow types $A \to B$). Such generalized pairs are called *dependent sums* where the type of the second component can depend on the first. Dependent sum types, denoted by $\Sigma x : A.\ B$, are also called Sigma-types. The typing rule of dependent sums is as follows:

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[x \mapsto e_1]}{\Gamma \vdash \mathbf{pack}\,[e_1, e_2]\,\mathbf{as}\,(\Sigma x : A.\ B) : (\Sigma x : A.\ B)}$$

where **pack** is the constructor of dependent sums. There are two forms of dependent sums, namely *weak* and *strong* sums. The two forms are distinguished by their destructors [Schmidt 1994].

2.2.1 Weak Sums

Weak sums are eliminated by *weak pattern matching*, i.e. the **unpack** operator, which is conducted within a particular scope and the pattern variables x and y cannot escape this scope. The typing rule for **unpack** is as follows:

$$\frac{\Gamma \vdash e_1 : \Sigma x : A.\ B \qquad \Gamma, x : A, y : B \vdash e_2 : C \qquad \Gamma \vdash C : \star}{\Gamma \vdash \mathbf{unpack}\ e_1\ \mathbf{as}\ [x, y]\ \mathbf{in}\ e_2 : C}$$

Notice that both x and y cannot occur free in C, since C is a well-formed type under the context Γ without x or y. Weak sum types can also be encoded with Pi-types in a similar way to Church-encoding existential types in System F [Pierce 2002]:

$$\begin{array}{rcll} \Sigma x : A.\ B & \triangleq & \Pi z : \star.\ (\Pi x : A.\ B \to z) \to z & z \text{ fresh} \\ \mathbf{pack}\ [e_1, e_2]\ \mathbf{as}\ \Sigma x : A.\ B & \triangleq & \lambda z : \star.\ \lambda f : (\Pi x : A.\ B \to z).\ f\ e_1\ e_2 & z \text{ fresh} \\ \mathbf{unpack}\ e\ \mathbf{as}\ [x, y]\ \mathbf{in}\ e' & \triangleq & e\ C\ (\lambda x : A.\ \lambda y : B.\ e') & x, y \notin \mathsf{FV}(C) \end{array}$$

where z is fresh such that $z \notin \mathsf{FV}(\Pi x : A.\ B)$. Note that C is the type of e'. A and B can be derived from the type of e, i.e., $\Sigma x : A.\ B$.

2.2.2 Strong Sums

Strong sums are eliminated by *projections* similarly to non-dependent pairs, which directly extract the first or second component without any scoping restriction. In particular, the type of the second projection can refer to the first. The standard typing rules for the first and second projection are as follows:

$$\frac{\Gamma \vdash e : \Sigma x : A.\ B}{\Gamma \vdash e.1 : A} \qquad \frac{\Gamma \vdash e : \Sigma x : A.\ B}{\Gamma \vdash e.2 : B[x \mapsto e.1]}$$

Note that x does not show (as a free variable) in the typing result of the second projection $e.2$, since all occurrences of x are replaced by the first projection $e.1$.

2.2.3 Comparison of Weak and Strong Sums

The projection operators of strong sums can be more expressive than the **unpack** operators of weak sums. For example, recall the encoding of abstract integer sets using dependent sums shown in Section 1.3.3. In Scala, we can write a generic function `f` on `Set`:

```
def f(s: Set) = s.member(3, s.insert(3, s.empty()))
```

This function is encodable by the **unpack** operator:

$$f = \lambda s : Set.\ \textbf{unpack}\ s\ \textbf{as}\ [\,T, s'\,]\ \textbf{in}\ s'.member\ 3\ (s'.insert\ 3\ s'.empty)$$

It can also be encoded using projections:

$$f = \lambda s : Set.\ (s.2).member\ 3\ ((s.2).insert\ 3\ (s.2).empty)$$

where the pattern variable s' is replaced by a direct second projection $s.2$. However, consider another generic function `g` that simply returns an empty set:

```
def g(s: Set) = s.empty()
```

We cannot encode `g` using **unpack** as follows:

$$g = \lambda s : Set.\ \textbf{unpack}\ s\ \textbf{as}\ [\,T, s'\,]\ \textbf{in}\ s'.empty \quad \text{-- ill-typed}$$

because $s'.empty$ has type T which violates the typing rule of **unpack**. The type of the sub-term in an **unpack** should not refer to pattern variables. In contrast, we can encode `g` with the second projection:

$$g = \lambda s : Set.\ (s.2).empty$$

The type of g is $\Pi(s : Set).\ (s.1)$ which is well-formed by using the first projection to represent the abstract type member.

Nonetheless, weak and strong sums both have their applications. Weak sums are useful for data abstraction [Mitchell and Plotkin 1988; Pierce 2002], while strong sums are suitable for representing modular structures [MacQueen 1986]. It usually takes less effort to support weak sums, since they are essentially polymorphic existential types which are encodable by Pi-types in traditional dependently typed systems, e.g., the calculus of constructions [Coquand and Huet 1988]. On the contrary, strong sums are beyond the expressiveness of the plain calculus of constructions and cannot be Church-encoded with Pi-types [Cardelli 1986b].

2.3 Iso-Recursive Types

Recursive types are types that can refer to themselves. For example in Haskell, an integer list type $IntList$ is recursively defined:

$$\textbf{data}\ IntList = Nil \mid Cons\ Int\ IntList$$

The definition indicates that an $IntList$ can be either an empty list (Nil) or a concatenation of a single integer (Int) and another integer list ($IntList$). Similarly, $IntList$ can be defined as a class in Java, which can be viewed as a recursive type:

```
class IntList {
    Integer value;
    IntList next;
}
```

In type theory, a recursive type is denoted by $\mu x : A.\ B$. The μ-operator is an explicit recursion operator. The binder x refers to the whole type itself, i.e., $x = \mu x : A.\ B$, and can show up in the body type B. The previous $IntList$ example can also be written with μ as follows [Pierce 2002]:

$$IntList = \mu x : \star.\ \{\mathit{Nil} : \mathit{Unit}, \mathit{Cons} : \mathit{Int}\ \times\ x\}$$

The binder x refers to $IntList$ itself, which is a type, i.e., has kind $\star$. The body is a record type (i.e. a named pair) consisting of two components Nil and $Cons$. The empty list case Nil is a singleton of $Unit$ type. The concatenation case $Cons$ is a pair consisting of a number and another list whose type is x, equivalently $IntList$.

2.3.1 Iso-Recursive versus Equi-Recursive Types

Recursive types can represent infinite structures by constantly expanding the body. Given $\mu x : A.\ B$, one can expand the body type B by *unfolding*, an operation that replaces all occurrences of x in B by the whole μ-term itself:

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

Such expansion/unfolding operation of recursive types can be performed either implicitly or explicitly. Accordingly, there are two ways of formalizing recursive types in the literature, namely the *equi-recursive* and *iso-recursive* approach [Crary et al. 1999]. Recursive types using such two approaches are called equi-recursive and iso-recursive types.

Equi-Recursive Types. In the equi-recursive approach, the recursive type and its unfolded type are *definitionally equal*, i.e., both types can be used interchangeably in any context. Thus, we have the following equivalence relation:

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

The benefit is that the equi-recursive treatment naturally allows type expressions to be infinite as the unfolding operation happens automatically. However, it is difficult to type-check equi-recursive types, since the type-checking algorithm usually does not work directly with infinite types [Pierce 2002].

Iso-Recursive Types. Another approach, the iso-recursive approach, treats a recursive type and its unfolding as different types, but *isomorphic* ones. One cannot replace a recursive type with its unfolded form or vice versa. Instead, the unfolding or folding (the opposite of unfolding) operation is triggered *explicitly* by two operators, unfold and fold, respectively. For example, 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:

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

where fold and unfold map back and forth between the original and unfolded form. Formally, we have the following standard typing rules of iso-recursive types [Pierce 2002]:

$$\frac{\Gamma \vdash e_1 : (\mu x : A.\ B)}{\Gamma \vdash \mathsf{unfold}\ e_1 : B[x \mapsto \mu x : A.\ B]} \qquad \frac{\Gamma \vdash (\mu x : A.\ B) : \star \quad \Gamma \vdash e_2 : B[x \mapsto \mu x : A.\ B]}{\Gamma \vdash \mathsf{fold}\ [\mu x : A.\ B] e_2 : (\mu x : A.\ B)}$$

The isomorphism between types of e_1 and e_2 is witnessed by fold and unfold:

$$\mu x : A.\ B \underset{\mathsf{fold}\ [\mu x : A.\ B]}{\overset{\mathsf{unfold}}{\rightleftharpoons}} B[x \mapsto \mu x : A.\ B]$$

The iso-recursive treatment makes it somewhat less convenient and intuitive to use recursive types, since one needs to explicitly add unfold and fold instructions whenever conversions of recursive types are involved. On the other hand, type-checking iso-recursive types is much easier than equi-recursive ones since the typing rules of unfold and fold are syntax-directed.

2.3.2 Iso-Recursive Types in Haskell

For practicality reasons, many programming languages implement recursive types using the iso-recursive approach instead of the equi-recursive approach. For example, algebraic datatypes in Haskell [Marlow et al. 2010] and ML [Milner et al. 1990] implicitly use iso-recursive types for recursively defined structures, such as the previous *IntList* example. Similarly, Java implicitly introduces a recursive type for the class definition and the method call of objects involves an implicit unfold operation [Pierce 2002].

In Haskell, one cannot directly define a recursive type using the **type** synonym:

$$\mathbf{type}\ Bad = Int \to Bad$$

The **type** synonym treats the types at two sides of $=$ as equal, similarly to the equi-recursive approach, which is however not supported in Haskell. Instead, one needs to use **newtype** (or **data**) to wrap the unfolded form (i.e. the right-hand side type) inside a constructor:

$$\mathbf{newtype}\ Good = MkGood\ (Int \to Good)$$

It is also possible to use **newtype** to define a generic fixpoint combinator for types:

$$\mathbf{newtype}\ Fix\ f = Fold\ \{\ unfold :: f\ (Fix\ f)\ \}$$

The type Fix is a type-level fixpoint using the iso-recursive approach. Its constructor $Fold$ forms a recursive type from function f and destructor $unfold$ forces the computation to obtain the unfolded form $f\ (Fix\ f)$. Such combinator can be used to model the recursion in datatypes and define them in a modular fashion, as illustrated in the *Data Types à la Carte* paper by Swierstra [2008].

2.4 Subtyping

Subtyping plays an important role in object-oriented programming (OOP) languages such Java to model *polymorphism*. Subtyping is a relation between two types, denoted by $A \leqslant B$, where A is called a *subtype* of B, and B is called a *supertype* of A. The relation means that whenever the context requires a term of type B, it can accept a term of type A. For example, in many languages such as C/C++, the integer type can be considered as a subtype of a floating-point number type, i.e. `int` $\leqslant$ `float`. Whenever one needs a `float`, an `int` can be used, but not vice versa. Such interpretation of subtyping is often called the *principle of safe substitution* [Pierce 2002]. To connect typing with the subtyping relation, we need a new typing rule called the *subsumption* rule:

$$\frac{\Gamma \vdash e : A \qquad A \leqslant B}{\Gamma \vdash e : B}$$

This rule indicates that any term of type A is also a term of type B, which complies with the interpretation of subtyping.

2.4.1 Important Subtyping Rules

We introduce several important subtyping rules that can be found in many languages.

Reflexivity and Transitivity. First, the subtyping relation should satisfy two basic properties: *reflexivity* and *transitivity*, i.e., subtyping should be a reflexive relation by the following rule:

$$A \leqslant A$$

and a transitive relation by the following rule:

$$\frac{A \leqslant B \qquad B \leqslant C}{A \leqslant C}$$

Notice that the transitivity rule is not *algorithmic* in the sense that one needs to guess the type B when applying the transitivity rule for $A \leqslant C$. When implementing a subtyping algorithm, one usually needs to eliminate the transitivity rule and prove that it is admissible from other subtyping rules [Pierce 2002; Pierce and Steffen 1997].

Function Types. In higher-order languages, functions are first-class values and can be passed as arguments. Thus, we need a subtyping rule between function types:

$$\frac{B_1 \leqslant A_1 \qquad A_2 \leqslant B_2}{A_1 \to A_2 \leqslant B_1 \to B_2}$$

Note that the subtype relation of the argument types is reversed. We call it *contravariant*. And the subtype relation has the same order for the result types as the function types. We call such relation *covariant*. The reason is that if a function has type $A_1 \to A_2$, it can take any arguments of B_1 which is a subtype of A_1 and returns terms of type A_2 which can be treated as having its supertype B_2. Thus, we can treat any function of type $A_1 \to A_2$ as having type $B_1 \to B_2$.

Record Types. Many languages support *records*, which are collections of named fields. The record type is denoted by $\{l_1 : T_1, \cdots, l_n : T_n\}$ where l_i ranges over names. There are three typing rules for record types [Pierce 2002]:

- The width subtyping rule:

$$\{l_1 : T_1, \cdots, l_n : T_n, \cdots, l_{n+k} : T_{n+k}\} \leqslant \{l_1 : T_1, \cdots, l_n : T_n\}$$

  which indicates that record types are subtypes if they have more fields at the end. Since every field showing in the supertype also shows in the subtype, any operation accepted by the supertype can be supported by the subtype.

- The depth subtyping rule:

$$\frac{S_i \leqslant T_i \qquad \forall i, 1 \leqslant i \leqslant n}{\{l_1 : S_1, \cdots, l_n : S_n\} \leqslant \{l_1 : T_1, \cdots, l_n : T_n\}}$$

  which indicates types of each corresponding fields are in the subtyping relation.

- The permutation subtyping rule:

$$\frac{\{\overline{l'_i : T'_i}\} \text{ is a permutation of } \{\overline{l_i : T_i}\}}{\{l'_1 : T'_1, \cdots, l'_n : T'_n\} \leqslant \{l_1 : T_1, \cdots, l_n : T_n\}}$$

  which indicates that the order of fields does not matter.

Top Types. Finally, in some languages, there exists a type which is a supertype of any type. We call it the *top type*, denoted by $\top$. We need the following subtyping rule for the top type:

$$A \leqslant \top$$

Many OOP languages have the top type which is usually the universal base class, e.g., `Object` in Java and `Any` in Scala.

2.4.2 Bounded Quantification

So far we have introduced the subtyping relation as a judgment $A \leqslant B$. Subtyping can also be used with a quantifier to require the binder to be a subtype of a specific upper bound. We call this *bounded quantification*. For example, System $F_{\leqslant}$ [Cardelli et al. 1994] is an extension to System F with subtyping and bounded quantification. The specification of $F_{\leqslant}$ is shown in Figure 2.3. In $F_{\leqslant}$, we have one extra type abstraction with an upper bound:

$$\Lambda X \leqslant T.\ t$$

where the binder X needs to be a subtype of type T. Correspondingly, the abstraction has a bounded universal type:

$$(\Lambda X \leqslant T.\ t) : (\forall X \leqslant T.\ T')$$

(Syntax)

$$\begin{array}{lrcl} \text{Types} & T, U & ::= & X \mid \top \mid T_1 \to T_2 \mid \forall X \leqslant U.\ T \\ \text{Terms} & t & ::= & x \mid \lambda x : T.\ t \mid t_1\ t_2 \mid \Lambda X \leqslant T.\ t \mid t[T] \\ \text{Contexts} & \Delta & ::= & \varnothing \mid \Delta, x : T \mid \Delta, X \leqslant T \end{array}$$

$\boxed{\Delta \vdash T}$ *(Type Well-formedness)*

FWT-Top
$$\frac{\vdash \Delta}{\Delta \vdash \top}$$

FWT-Var
$$\frac{\vdash \Delta \qquad X \leqslant U \in \Delta}{\Delta \vdash X}$$

FWT-Arrow
$$\frac{\Delta \vdash T_1 \qquad \Delta \vdash T_2}{\Delta \vdash T_1 \to T_2}$$

FWT-Forall
$$\frac{\Delta \vdash T_1 \qquad \Delta, X \leqslant T_1 \vdash T_2}{\Delta \vdash \forall X \leqslant T_1.\ T_2}$$

$\boxed{\vdash \Delta}$ *(Context Well-formedness)*

FW-Empty
$$\frac{}{\vdash \varnothing}$$

FW-Type
$$\frac{\vdash \Delta \qquad \Delta \vdash T}{\vdash \Delta, X \leqslant T}$$

FW-Term
$$\frac{\vdash \Delta \qquad \Delta \vdash T}{\vdash \Delta, X : T}$$

$\boxed{\Delta \vdash T \leqslant U}$ *(Subtyping)*

FS-Top
$$\frac{\Delta \vdash T}{\Delta \vdash T \leqslant \top}$$

FS-VarRefl
$$\frac{\Delta \vdash X}{\Delta \vdash X \leqslant X}$$

FS-VarTrans
$$\frac{X \leqslant T_1 \in \Delta \qquad \Delta \vdash T_1 \leqslant T_2}{\Delta \vdash X \leqslant T_2}$$

FS-Arrow
$$\frac{\Delta \vdash T_1 \leqslant U_1 \qquad \Delta \vdash U_2 \leqslant T_2}{\Delta \vdash (U_1 \to U_2) \leqslant (T_1 \to T_2)}$$

FS-Forall
$$\frac{\Delta, X \leqslant U \vdash T_1 \leqslant T_2}{\Delta \vdash (\forall X \leqslant U.\ T_1) \leqslant (\forall X \leqslant U.\ T_2)}$$

$\boxed{\Delta \vdash t : T}$ *(Typing)*

FT-Var
$$\frac{\vdash \Delta \qquad x : T \in \Delta}{\Delta \vdash x : T}$$

FT-Abs
$$\frac{\Delta, x : T_1 \vdash t : T_2}{\Delta \vdash (\lambda x : T_1.\ t) : T_1 \to T_2}$$

FT-App
$$\frac{\Delta \vdash t_1 : T_1 \to T_2 \qquad \Delta \vdash t_2 : T_1}{\Delta \vdash t_1\ t_2 : T_2}$$

FT-TAbs
$$\frac{\Delta, X \leqslant T_1 \vdash t : T_2}{\Delta \vdash (\Lambda X \leqslant T_1.\ t) : (\forall X \leqslant T_1.\ T_2)}$$

FT-TApp
$$\frac{\Delta \vdash t : (\forall X \leqslant U_1.\ U_2) \qquad \Delta \vdash T \leqslant U_1}{\Delta \vdash t[T] : U_2[X \mapsto T]}$$

FT-Sub
$$\frac{\Delta \vdash t : T \qquad \Delta \vdash T \leqslant U}{\Delta \vdash t : U}$$

Figure 2.3. Specification of System $F_{\leqslant}$

The original unbounded quantification of System F can be viewed as a special case of top-bounded quantification:

$$\forall X.\ T \triangleq \forall X \leqslant \top.\ T$$

Variants of System $F_{\leqslant}$. There are two variants of System $F_{\leqslant}$, the *Kernel Fun* variant [Cardelli and Wegner 1985] and the *Full* variant [Cardelli et al. 1994]. Figure 2.3 shows the Kernel Fun variant. The only difference between two variants is the subtyping rule of bounded universal

types. In the Kernel Fun variant, two bounded universal types can be compared only when their bounds are identical, i.e., invariant:

$$\frac{\Delta, X \leqslant U \vdash T_1 \leqslant T_2}{\Delta \vdash (\forall X \leqslant U.\ T_1) \leqslant (\forall X \leqslant U.\ T_2)}$$

In the Full variant, the bounds are contravariant, hence not required to be equal:

$$\frac{\Delta \vdash U_2 \leqslant U_1 \qquad \Delta, X \leqslant U_2 \vdash T_1 \leqslant T_2}{\Delta \vdash (\forall X \leqslant U_1.\ T_1) \leqslant (\forall X \leqslant U_2.\ T_2)}$$

The subtyping rule of the Full variant seems more natural, which follows the treatment of subtyping function types. However, this rule makes Full $F_{\leqslant}$ undecidable [Pierce 1992], meaning that the algorithm implementing subtyping rules of Full $F_{\leqslant}$ does not terminate for certain inputs. On the contrary, Kernel Fun $F_{\leqslant}$ is decidable by using the restricted invariant subtyping rule. Thus, from the perspective of implementation, the Kernel Fun variant is more interesting since it can have a decidable subtyping algorithm.

2.5 Path-Dependent Types

Scala [Odersky et al. 2004] supports *path-dependent types* [Odersky et al. 2003; Amin et al. 2014], a weaker form of dependent types that only allow types to depend on paths. The syntax of path-dependent types (i.e. paths) can be formally defined as follows [Odersky et al. 2003]:

$$p ::= x \mid p.L$$

A path p is a variable x followed by a sequence of selections, e.g., $x.L_1.\ \cdots\ .L_n$. In Dependent Object Type (DOT) [Rompf and Amin 2016; Amin et al. 2016], a calculus that captures core features of Scala, paths are further simplified to only allow single selections:

$$p ::= x.L$$

Meanwhile, paths are valid types:

$$T ::= \cdots \mid p$$

Thus, paths can appear with other types such as universal types:

$$\forall (x : T).\ (x.L)$$

A concrete example is the *dependent method type* of the generic method `g` from Section 2.2.3:

```
def g(s: Set) = s.empty()
g: (s: Set)s.T
```

where the result type `s.T` can refer to the parameter `s` of the method.

Comparison with Full Dependent Types. The need of path-dependent types arises from accessing abstract type members of traits which involves some forms of value dependency (see also Section 1.3.3). Scala takes a conservative approach by only supporting path dependency

and avoids the need of supporting more general forms of dependent types. There are two major differences between path-dependent types and full dependent types from traditional dependently typed calculi (e.g. the calculus of constructions [Coquand and Huet 1988]).

First, Pi-types can support dependency on any terms instead of only path dependency, e.g., a vector type $Vec\ n$ where n can be any natural number. Such vector types are not expressible with path-dependent types. Moreover, with strong sums, Pi-types along with projections can simulate path-dependent types. For example, in Section 2.2.3, the function `g` can be encoded with the second projection and its type is a Pi-type with the first-projection: $\Pi(s : Set).\ (s.1)$.

Second, traditional dependent types usually support arbitrary computations at the type level, e.g., $Vec\ (1 + 2) = Vec\ 3$. In contrast, path-dependent types only support simple forms of type conversions, such as instantiating abstract type members with actual types. Such type conversions can be done with simple type equality relations. For example, if we apply an implementation of `Set` to `g` such that `T=List[Int]`, we will have:

```
g(new Set {type T = List[Int]; ... }): List[Int]   // excerpted
```

The original result type, i.e., the abstract type `s.T`, is instantiated with the actual type `List[Int]` according to the type equality `T=List[Int]`.

Part I:

Pure Iso-Type Systems

CHAPTER 3

Overview of Iso-Types

In this chapter, we give an overview of iso-types and informally introduce key features of Pure Iso-Type Systems (PITS). PITS enables the combination of unified syntax and general recursion, while retains decidable type checking at the same time. The main source of inspiration for the design of PITS comes from *iso-recursive types* [Crary et al. 1999; Pierce 2002], which we briefly introduced in Section 1.4.1. Instead of an *implicit* type conversion (employed by PTS), PITS provides a generalization of *iso-recursive types* called *iso-types*. In PITS, not only folding/unfolding of recursive types is explicitly controlled by term level constructs, but also any other type-level computation (including beta reduction/expansion). This is somewhat similar to how datatypes or newtypes are used in (classic) Haskell to explicitly trigger type-level computation for both recursive and non-recursive types.

There is an analogy to language designs with *equi-recursive* types and *iso-recursive* types. With equi-recursive types, type-level recursion is implicitly folded/unfolded, which makes establishing decidability of type-checking much more difficult. In iso-recursive designs, the idea is to trade some convenience by a simple way to ensure decidability. Similarly, we view the design of traditional dependently typed calculi, such as 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 the normalization property [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.

It is worth emphasizing the goal of PITS is to show the benefits of PTS-style unified syntax in terms of *economy of concepts* for more traditional programming language designs, but not to use unified syntax to express computationally intensive type-level programs. Traditional functional languages only require basic use of type-level computation to support common features like parametrized algebraic data types, iso-recursive types or purely functional objects [Pierce and Turner 1994]. These features require only one or a small number of finite steps of type reductions/expansions where the iso-type approach fits well. In Section 3.2, we give examples to illustrate how such features of modern functional languages can be encoded with iso-types in PITS.

3.1 Motivation and Overview

3.1.1 Implicit Type Conversion in Pure Type Systems

The typing rules for PTS contain a conversion rule [Barendregt 1991] (see also Section 2.1.1):

$$\frac{\Gamma \vdash e : A \qquad A =_\beta B}{\Gamma \vdash e : B}$$

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:

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

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

Decidability of Type Checking and Strong Normalization. While the conversion rule in 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 loop, where loop stands for any diverging computation. If we type check $(\lambda x : Int.\ x)\ z$ under the normal typing rules of PTS, the type checker would get stuck as it tries to do beta equality on two terms: Int and loop, where the latter is non-terminating.

3.1.2 Newtypes: 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, as introduced in Section 2.3.2, 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 Id defined by

$$\textbf{newtype}\ Id\ a = MkId\ \{runId :: a\}$$

can be viewed as a type-level identity function and $Id\ a$ is isomorphic to type a. To convert the type back and forth between a and $Id\ a$, one needs to explicitly use the constructor $MkId$ or the destructor $runId$:

$$MkId :: a \to Id\ a$$
$$runId :: Id\ a \to a$$

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 the monadic programming style that is available in Haskell, or modular interpreters enabled by approaches such as Datatypes à la Carte [Swierstra 2008].

3.1.3 Iso-Types: Explicit Type Conversion in PITS

PITS iso-types have strong similarities with the explicit conversion mechanism found in languages like Haskell, and iso-recursive types. 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.

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: $\mathsf{cast}_{\downarrow}$ (one-step reduction) and $\mathsf{cast}_{\uparrow}$ (one-step expansion). One benefit of this approach is that decidability of type checking is no longer coupled with strong normalization of the calculus. Another potential benefit 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. Although we do not explore the later point in this thesis, Jones' work on type-inference for higher-kinded types in Haskell [Jones 1993] seems to back up this idea.

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

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

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

Expansion. The dual operation of $\mathsf{cast}_{\downarrow}$ is $\mathsf{cast}_{\uparrow}$, which allows a type conversion provided that the resulting type is an *expansion* of the original type of the term. To explain the use of $\mathsf{cast}_{\uparrow}$, let us revisit the example from Section 3.1.1. We cannot apply f to 3 without the conversion rule. Instead, we can use $\mathsf{cast}_{\uparrow}$ to expand the type of 3:

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

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

One-Step. The cast 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 $e : (\lambda x : \star.\ \lambda y : \star.\ x)\ Int\ Bool$. Given $g = \lambda y : Int.\ y$, the expression $g\ (\mathsf{cast}_{\downarrow}\ e)$ is ill-typed because $\mathsf{cast}_{\downarrow}\ e$ has type $(\lambda y : \star.\ Int)\ Bool$, which is not syntactically equal to Int. Thus, we need another $\mathsf{cast}_{\downarrow}$:

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

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

Analogy to Newtypes in Haskell. By using cast operators, we can model the Haskell example shown in Section 3.1.2:

$$\begin{array}{l} Id = \lambda x : \star.\ x \\ \mathsf{cast}_{\uparrow}\ [Id\ a] : a \to Id\ a \\ \mathsf{cast}_{\downarrow}\ \ \ \ \ \ \ \ : Id\ a \to a \end{array}$$

$\mathsf{cast}_{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ are analogous to the datatype constructor $MkId$ and destructor $runId$, respectively. The difference is that PITS directly supports type-level lambdas without the need of using **newtype**, though PITS only has alpha equality without implicit beta conversion. In some sense, type-level lambdas in PITS are *non-computational* as **newtype**-style "type functions" in Haskell. Nevertheless, we can still trigger type-level computation by casts, similarly to **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 3.1.1. This time the type checker will not get stuck when type checking $(\lambda x : Int.\ x)\ z$. This is because, in PITS, the type checker only performs syntactic comparison between Int and loop, instead of beta equality. Thus it rejects the above application as ill-typed. Indeed it is impossible to type check such application even with the use of $\mathsf{cast}_{\uparrow}$ and/or $\mathsf{cast}_{\downarrow}$: one would need to write infinite number of $\mathsf{cast}_{\downarrow}$'s to make the type checker loop forever (e.g., $(\lambda x : Int.\ x)(\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow} \ldots z))$). But it is impossible to write such program in practice which has an infinite length and cannot be stored with a finite capacity.

Variants of Casts. A reduction relation is used in cast operators to convert types. We study *three* possible reduction relations: *call-by-name* reduction, *call-by-value* reduction and *full* reduction. Call-by-name and call-by-value reduction cannot reduce sub-terms at certain positions (e.g., inside λ or Π binders), while full reduction can reduce sub-terms at any position. We also create three variants of PITS for each variant of casts. Specifically, full PITS uses a *decidable parallel reduction* relation with full cast operators $\mathsf{cast}_{\Uparrow}$ and $\mathsf{cast}_{\Downarrow}$. 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 $\lambda x : \mathit{Int}.\ (1 + 1)$ to $\lambda x : \mathit{Int}.\ 2$. Full PITS is more expressive but results in a more complicated metatheory (see Section 4.3). Note that when generally referring to PITS, we do not specify the reduction strategy, which could be any variant.

3.1.4 General Recursion

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

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

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

We treat the μ operator as a *fixpoint*, which evaluates $\mu x : A.\ e$ 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., $\mathit{fact}\ 3$ produces 6 as expected (see Section 4.1.4).

Recursive Types. The same μ 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 (see also Section 2.3.1). In call-by-name PITS, such isomorphism is witnessed by $\mathsf{cast}_{\uparrow}$ and $\mathsf{cast}_{\downarrow}$. In fact, $\mathsf{cast}_{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ *generalize* fold and unfold: they can convert any types, not just recursive types, as we shall see in the example of encoding parametrized datatypes in Section 3.2.

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

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

3.2 Iso-Types by Example

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. We focus on common features available in traditional functional languages and some interesting type-level features, but not the *full power* of dependent types. **Fun** is not logically consistent due to the "type-in-type" axiom. In this section, we briefly introduce the implementation of **Fun** and present interesting examples including algebraic datatypes (Section 3.2.1), higher-kinded types (Section 3.2.2), datatype promotion (Section 3.2.2), high-order abstract syntax (Section 3.2.2) and object encodings (Section 3.2.3). All of those examples work in the 3 variants of PITS. We also discuss one final example on dependently typed vectors (Section 3.2.4) that only works with parallel reduction. All examples can run in the prototype interpreter and compiler (see Section 1.5).

3.2.1 Fun Implementation

Fun is built on top of a call-by-name PITS variant called λI [Yang et al. 2016] and provides surface language constructs for algebraic datatypes and pattern matching. λI has the same PTS specification as $\lambda\star$ (see Section 2.1.2), where $\mathcal{S} = \{\star\}$, $\mathcal{A} = \{(\star, \star)\}$ and $\mathcal{R} = \{(\star, \star, \star)\}$. Algebraic datatypes and pattern matching in **Fun** are implemented using Scott encodings [Mogensen 1992], which can be later desugared into PITS (λI) 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 λI terms and then type-checked using PITS typing rules. The type-checked λI terms can be evaluated directly by interpreter or compiled to JavaScript or Haskell code.

Encoding Parametrized Algebraic Datatypes. We give an example of encoding parametrized algebraic datatypes in PITS via the μ-operator and *call-by-name* casts. Importantly we should note that having iso-recursive types alone (and alpha equality) would be insufficient to encode parametrized types: the generalization afforded by iso-types is needed here.

In **Fun** we can define a *polymorphic list* as

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

This **Fun** definition is translated into PITS using a Scott encoding [Mogensen 1992] of datatypes:

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

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

We have two notable remarks from the example above. 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 3.2.4).

3.2.2 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 Π-types $\Pi x : A.B$ in the following text.

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_ω [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*:

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

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:

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

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

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

Since Maybe has kind $\star \to \star$, it is legal to apply Func to Maybe.

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:

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

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:

$$Fork\ Z\ 1\ (Empty\ (S\ Z))\ (Empty\ (S\ Z))$$

If we accidentally write the wrong depth, for example:

$$Fork\ Z\ 1\ (Empty\ (S\ Z))\ (Empty\ Z)$$

The above will fail to pass type checking.

Higher-order Abstract Syntax. *Higher-order abstract syntax* [Pfenning and 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:

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

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 2007b] 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 [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; texts after "--" are comments):

$$\begin{array}{l}\textbf{data}\ Value = VI\ Int \mid VF\ (Value \to Value);\\ \textbf{data}\ Eval = Ev\ \{\, eval' : Exp \to Value, reify' : Value \to Exp\,\};\\ \textbf{defrec}\ ev : Eval =\\ \quad Ev\ (\lambda e : Exp.\quad \textbf{case}\ e\ \textbf{of}\ \ldots \quad \text{-- excerpted}\\ \qquad\qquad\qquad\qquad \mid\ Lam\ f \Rightarrow VF\ (\lambda e' : Value.\ eval'\ ev\ (f\ (reify'\ ev\ e')))\\ \qquad (\lambda v : Value.\ \textbf{case}\ v\ \textbf{of}\ \ldots \quad \text{-- excerpted}\\ \qquad\qquad\qquad\qquad \mid\ VF\ f \Rightarrow Lam\ (\lambda e' : Exp.\ reify'\ ev\ (f\ (eval'\ ev\ e')));\\ \textbf{def}\ eval : Exp \to Value = eval'\ ev;\end{array}$$

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 $Value \to Exp$ 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:

$$\begin{array}{l}\textbf{def}\ show = \lambda v : Value.\ \textbf{case}\ v\ \textbf{of}\ VI\ n \Rightarrow n;\\ \textbf{def}\ expr = App\ (Lam\ (\lambda f : Exp.\ App\ f\ (Num\ 42)))\ (Lam\ (\lambda g : Exp.\ g));\\ show\ (eval\ expr)\quad \text{-- returns 42}\end{array}$$

3.2.3 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. We present an example of existential object encodings [Pierce and Turner 1994] in **Fun**, which originally requires System F_ω. Note that the example here relies on the facility of algebraic datatypes and records in **Fun**. It does not involve message passing due to the lack of subtyping in PITS. A more complete example of object encodings will be shown in Section 5.1.5. First, we encode the existential type and its constructor in **Fun** by standard Church encoding [Pierce 2002] using the universal type (i.e. Π-type):

$$\begin{array}{l}\textbf{def}\ Ex\ \ = \lambda P : \star \to \star.\ (z : \star) \to ((x : \star) \to P\ x \to z) \to z;\\ \textbf{def}\ pack = \lambda P : \star \to \star.\ \lambda e_1 : \star.\ \lambda e_2 : P\ e_1.\\ \qquad\qquad \mathsf{cast}_\uparrow\ [Ex\ P]\ (\lambda z : \star.\ \lambda f : (x : \star) \to P\ x \to z.\ f\ e_1\ e_2);\end{array}$$

where $pack$ is the constructor to build an existential package, which is similar to the encoding of $pack$ operators for weak sums as introduced in Section 2.2.1. Thus, we can encode an existential type $\exists x.\ A$ as $Ex\ (\lambda x : \star.\ A)$ in **Fun**. The object type operator Obj can be encoded as follows:

$$\begin{array}{l}\textbf{data}\ Pair\ (A : \star)\ (B : \star) = MkPair\ \{fst : A, snd : B\};\\ \textbf{def}\ Obj = \lambda I : \star \to \star.\ Ex\ (\lambda X : \star.\ Pair\ X\ (X \to I\ X));\end{array}$$

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 $X \to I\ X$). For a concrete example of objects, we use the interface of cell objects [Bruce et al. 1999]:

$$\textbf{data}\ Cell\ (X : \star) = MkCell\ \{get : Int, set : Int \to X, bump : X\};$$

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:

$$\begin{array}{l}\textbf{data}\ Var\ \ = MkVar\ \{getVar : Int\};\\ \textbf{def}\ CellT = \lambda X : \star.\ Pair\ X\ (X \to Cell\ X);\\ \textbf{def}\ pair\ \ = MkPair\ Var\ (Var \to Cell\ Var)\\ \qquad\qquad (MkVar\ 0)\qquad\qquad \text{-- Initial state}\end{array}$$

$$
\begin{array}{lll}
(\lambda s : Var.\ MkCell\ Var & \text{-- Methods} & \\
\quad (getVar\ s) & \text{-- Method } \textit{get} & \\
\quad (\lambda n : Int.MkVar\ n) & \text{-- Method } \textit{set} & \\
\quad (MkVar\ (getVar\ s + 1))); & \text{-- Method } \textit{bump} &
\end{array}
$$

$$\textbf{def}\ p \quad = pack\ CellT\ Var\ (\mathsf{cast}_{\uparrow}\ [CellT\ Var]\ pair);$$

$$\textbf{def}\ c \quad = \mathsf{cast}_{\uparrow}\ [Obj\ Cell]\ p;$$

The body of object c is an existential package p of type $\Sigma X.Pair\ X\ (X \to Cell\ X)$ built by the $pack$ operator. The first parameter of $pack$ is $CellT$ that represents the body of the existential type. The second parameter is the integer variable type Var which corresponds to the existential binder X. The third parameter has type $CellT\ Var$ which can be reduced to a pair type $Var \times (Var \to Cell\ Var)$ which is defined in the definition $pair$ constructed by $MkPair$. The first component of the $pair$ is the initial hidden state $MkVar\ 0$. The second component is a function containing three methods that are defined in a record by $MkCell$ and abstracted by the state variable s. The definition of the three methods follows the cell object interface $Cell$.

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

$$Obj\ Cell \hookrightarrow Ex\ (\lambda X : \star.\ Pair\ X\ (X \to Cell\ X)) = Ex\ CellT$$

We can use $\mathsf{cast}_{\uparrow}$ to do one-step type expansion for the package. Similarly, the second $\mathsf{cast}_{\uparrow}$ operator used in the third parameter of $pack$ converts the pair type into $CellT\ Var$. 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.

3.2.4 Fun with Full Reduction

So far all the examples can be encoded in **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 **def** bindings to top-level annotations without repeating them in λ-binders, e.g., we change $\textbf{def}\ f = \lambda x : Int.\ 1$ into

$$\textbf{def}\ f : Int \to Int = \lambda x.\ 1$$

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

$$
\begin{array}{l}
\textbf{data}\ Eq\ (k : \star)\ (a : k)\ (b : k) = \\
\quad Proof\ \{ subst : (f : k \to \star) \to f\ a \to f\ b \};
\end{array}
$$

Note that the definition requires *kind polymorphism*: the kind of types a and b is k, which is polymorphic and not limited to $\star$. For brevity, we use $a \equiv b$ to denote $Eq\ k\ a\ b$ by omitting the kind k. Then we can encode a GADT, for example, length-indexed list (or *vector*) as follows:

$$\begin{aligned}&\textbf{data}\ \mathit{Vec}\ (a : \star)\ (n : \mathit{Nat}) = \\ &\quad \mathit{Nil}\ (\mathit{Eq}\ \mathit{Nat}\ n\ \mathit{Z}) \\ &\quad \mid \mathit{Cons}\ (m : \mathit{Nat})\ (\mathit{Eq}\ \mathit{Nat}\ n\ (\mathit{S}\ m))\ a\ (\mathit{Vec}\ a\ m);\end{aligned}$$

However, it is difficult to use such encoding approach to express the injectivity of constructors [Cheney and Hinze 2003], e.g., deducing $n \equiv m$ from $S\ n \equiv S\ m$. It would be challenging to encode the tail function of a vector:

$$\begin{aligned}&\textbf{def}\ \mathit{tail} : (a : \star) \to (n : \mathit{Nat}) \to (v : \mathit{Vec}\ a\ (S\ n)) \to \mathit{Vec}\ a\ n = \\ &\quad \lambda a.\ \lambda n.\ \lambda v.\ \textbf{case}\ v\ \textbf{of} \\ &\qquad \mathit{Cons}\ m\ p\ x\ xs \Rightarrow xs; \quad \text{-- ill-typed}\end{aligned}$$

The case expression above is ill-typed: xs has the type $\mathit{Vec}\ a\ m$, but the function requires the case branch to return $\mathit{Vec}\ a\ n$. To convert xs to the correct type, we need to show $n \equiv m$. But the equality proof p has type $\mathit{Eq}\ \mathit{Nat}\ (S\ n)\ (S\ m)$, i.e., $S\ n \equiv S\ m$. Thus, the injectivity of constructor S is needed.

Fun incorporates two *full* cast operators ($\mathsf{cast}_{\Uparrow}$ and $\mathsf{cast}_{\Downarrow}$) from full PITS. With the power of full casts, we can "prove" the injectivity of S. We first define a *partial* function $\mathit{predNat}$ to destruct S:

$$\begin{aligned}&\textbf{def}\ \mathit{predNat} : \mathit{Nat} \to \mathit{Nat} = \\ &\quad \lambda n.\ \textbf{case}\ n\ \textbf{of}\ S\ m \Rightarrow m;\end{aligned}$$

Given $S\ n \equiv S\ m$, by congruence of equality, it is trivial to show $\mathit{predNat}\ (S\ n) \equiv \mathit{predNat}\ (S\ m)$. Noticing the reduction $\mathit{predNat}\ (S\ n) \hookrightarrow n$ holds, we can use a full cast operator $\mathsf{cast}_{\Downarrow}$ to reduce both sides of the equality to obtain $n \equiv m$:

$$\begin{aligned}&\textbf{def}\ \mathit{injNat} : (n : \mathit{Nat}) \to (m : \mathit{Nat}) \to \mathit{Eq}\ \mathit{Nat}\ (S\ n)\ (S\ m) \to \mathit{Eq}\ \mathit{Nat}\ n\ m = \\ &\quad \lambda n.\ \lambda m.\ \lambda p.\ \mathsf{cast}_{\Downarrow}\ (\mathit{lift}\ \mathit{Nat}\ \mathit{Nat}\ (S\ n)\ (S\ m)\ \mathit{predNat}\ p);\end{aligned}$$

The function lift (definition omitted) lifts the type of equality proof p from $S\ n \equiv S\ m$ to $\mathit{predNat}\ (S\ n) \equiv \mathit{predNat}\ (S\ m)$. Then $\mathsf{cast}_{\Downarrow}$ converts it to $\mathit{Eq}\ \mathit{Nat}\ n\ m$ (type annotations are omitted for brevity):

$$\begin{aligned}&p && : S\ n \equiv S\ m \\ &\mathit{lift}\ \mathit{predNat}\ p && : \mathit{predNat}\ (S\ n) \equiv \mathit{predNat}\ (S\ m) \\ &\mathsf{cast}_{\Downarrow}\ (\mathit{lift}\ \mathit{predNat}\ p) && : n \equiv m\end{aligned}$$

We can finally write a well-typed version of tail:

$$\begin{aligned}&\textbf{def}\ \mathit{castVec} : (a : \star) \to (n : \mathit{Nat}) \to (m : \mathit{Nat}) \to \\ &\qquad\qquad \mathit{Eq}\ \mathit{Nat}\ n\ m \to \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n = \ldots; \quad \text{-- excerpted} \\ &\textbf{def}\ \mathit{tail} : (a : \star) \to (n : \mathit{Nat}) \to (v : \mathit{Vec}\ a\ (S\ n)) \to \mathit{Vec}\ a\ n = \\ &\quad \lambda a.\ \lambda n.\ \lambda v.\ \textbf{case}\ v\ \textbf{of} \\ &\qquad \mathit{Cons}\ m\ p\ x\ xs \Rightarrow \mathit{castVec}\ a\ n\ m\ (\mathit{injNat}\ n\ m\ p)\ xs;\end{aligned}$$

where $castVec$ uses the proof $n \equiv m$ to convert xs from $Vec\ a\ m$ to $Vec\ a\ n$. Note that **Fun** is not logically consistent and does not check whether the proof is terminating. However, it is easy to see the injectivity proof $injNat$ from the example above is total — though $predNat$ is a partial function, it is always applied to numbers with the form $S\ n$.

CHAPTER 4

Pure Iso-Type Systems

This chapter formally presents Pure Iso-Type Systems (PITS), a family of calculi which employs unified syntax, supports general recursion and preserves decidable type-checking. PITS is comparable in simplicity to PTS. The main idea is to recover decidable type-checking through iso-types. In PITS, every type-level computation step is explicit and 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-kinded types [Girard 1972], or kind polymorphism [Yorgey et al. 2012], see also Section 3.2).

In this chapter, 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 (Section 4.1) uses weak-head call-by-name reduction, while *call-by-value* PITS (Section 4.2) 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[1] of reduction at both type and term level. While such consistency is easily ensured in a strongly normalizable calculus, as a term will always evaluate to the same normal form regardless of the reduction strategy, this it is not true for PITS which enables general recursion and loses strong normalization. For example, given term $(\lambda x : \mathit{Int}.\ 1)\ \bot$, where $\bot$ is any diverging computation, such term evaluates to 1 with call-by-name semantics, but diverges with call-by-value semantics. In call-by-name/value PITS, we can trivially guarantee that no invalid reasoning steps can happen due to mismatches with the evaluation strategy.

Unfortunately, both call-by-name and call-by-value reduction are not congruent for type equivalence (e.g. $\lambda x : \mathit{Int}.\ 1 + 1 \not\leftrightarrow \lambda x : \mathit{Int}.\ 2$), which loses some expressiveness in terms of type level computation. The third variant called *full* PITS (Section 4.3) uses *parallel reduction* for

[1] Note that the consistency of behavior is not *logical* consistency, but meant for reduction relations.

casts. Full PITS is more expressive than call-by-name/value PITS, and its type-level reduction is complete with respect to full beta-reduction employed by traditional PTS. Full PITS allows equating terms such as $Int \to Vec\ (1+1)$ and $Int \to Vec\ 2$ as equal, which is not possible in call-by-name/value PITS. The price to pay for this more expressive and congruent design is some additional complexity of the formalization, and the lack of consistency between term and type level reduction[2] (see Section 4.4.6). For all variants, type soundness and decidability of type-checking are proved.

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 a 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. Similarly, many previous calculi including PTS_f, a closely related calculus which is also based on PTS and explicit type conversions, rely on other variants of PTS to provide dynamic semantics and type-safety proofs. A detailed discussion of the trade-offs between the variants of PITS, as well as PTS_f, is given in Section 4.4.

4.1 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 4.1.1). 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.

4.1.1 Syntax

Fig. 4.1 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 *Sorts*, e.g., $\star$, $\Box$, 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 $A \to B$ as a syntactic sugar for $\Pi x : A.\ B$ if x does not occur free in B.

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

[2] For example, $(\lambda x : Int.\ 1+1)\ \bot$ is a non-terminating term under call-by-value reduction but can be reduced to $(\lambda x : Int.\ 2)\ \bot$ by parallel reduction.

$$
\begin{array}{lrcl}
\text{Expressions} & e, A, B & ::= & x \mid s \mid e_1\ e_2 \mid \lambda x : A.\ e \mid \Pi x : A.\ B \\
 & & \mid & \mu x : A.\ e \mid \mathsf{cast}_{\uparrow}\ [A]\ e \mid \mathsf{cast}_{\downarrow}\ e \\
\text{Values} & v & ::= & s \mid \lambda x : A.\ e \mid \Pi x : A.\ B \mid \mathsf{cast}_{\uparrow}\ [A]\ e \\
\text{Contexts} & \Gamma & ::= & \varnothing \mid \Gamma, x : A \\
\text{Syntactic Sugar} & & & \\
\quad A \to B & \triangleq & \Pi x : A.\ B & \text{where } x \notin \mathsf{FV}(B) \\
\quad \mathsf{cast}^n_{\uparrow}\ [A_1]\ e & \triangleq & \mathsf{cast}_{\uparrow}[A_1](\mathsf{cast}_{\uparrow}[A_2](\ldots(\mathsf{cast}_{\uparrow}\ [A_n]\ e)\ldots)) & \\
 & & & \text{where } A_1 \hookrightarrow A_2 \hookrightarrow \cdots \hookrightarrow A_n \\
\quad \mathsf{cast}^n_{\downarrow}\ e & \triangleq & \underbrace{\mathsf{cast}_{\downarrow}(\mathsf{cast}_{\downarrow}(\ldots(\mathsf{cast}_{\downarrow}}_{n}\ e)\ldots)) &
\end{array}
$$

Figure 4.1. Syntax of call-by-name PITS

the result type of one-step expansion for disambiguation, while $\mathsf{cast}_{\downarrow}$ does not, since the result type of one-step reduction can be uniquely determined as discussed in Section 4.1.5.

We use syntactic sugar $\mathsf{cast}^n_{\uparrow}$ and $\mathsf{cast}^n_{\downarrow}$ to denote n consecutive cast operators (see Fig. 4.1). 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 $\mathsf{cast}^n_{\uparrow}$ 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 4.1.2), the intermediate types can be uniquely determined and left out.

General Recursion. We use the μ-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 $Int \to Int$ or $s_1 \to s_2$.

4.1.2 Operational Semantics

Fig. 4.2 shows the small-step, *call-by-name* operational semantics. Three base cases include rule R-Beta for beta reduction, rule R-Mu for recursion unfolding 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 $\mathsf{cast}_{\downarrow}$ terms, respectively. Note that rule R-CastElim and rule R-Castdn do not overlap because in the former rule, the inner term of $\mathsf{cast}_{\downarrow}$ is a value (see Fig. 4.1), i.e., $\mathsf{cast}_{\uparrow}\ [A]\ e$. 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 λ-term and Π-term which are both defined as values. 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.

4.1.3 Typing

Fig. 4.3 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

$\boxed{e_1 \hookrightarrow e_2}$ *(Call-by-name Reduction)*

$$\textsc{R-Beta}\ \frac{}{(\lambda x : A.\ e_1)\ e_2 \hookrightarrow e_1[x \mapsto e_2]} \qquad \textsc{R-App}\ \frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \textsc{R-Mu}\ \frac{}{\mu x : A.\ e \hookrightarrow e[x \mapsto \mu x : A.\ e]}$$

$$\textsc{R-CastDn}\ \frac{e \hookrightarrow e'}{\mathsf{cast}_{\downarrow}\ e \hookrightarrow \mathsf{cast}_{\downarrow}\ e'} \qquad \textsc{R-CastElim}\ \frac{}{\mathsf{cast}_{\downarrow}\ (\mathsf{cast}_{\uparrow}\ [A]\ e) \hookrightarrow e}$$

Figure 4.2. Operational semantics of call-by-name PITS

$\boxed{\Gamma \vdash e : A}$ *(Typing of Call-by-name PITS)*

$$\textsc{T-Ax}\ \frac{\vdash \Gamma \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2} \qquad \textsc{T-Var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{T-Abs}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash e : B \qquad \Gamma, x : A \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash \lambda x : A.\ e : \Pi x : A.\ B}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash e_1 : \Pi x : A.\ B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B[x \mapsto e_2]} \qquad \textsc{T-Prod}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.\ B : s_3}$$

$$\textsc{T-Mu}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash e : A}{\Gamma \vdash \mu x : A.\ e : A} \qquad \textsc{T-CastUp}\ \frac{\Gamma \vdash B : s \qquad \Gamma \vdash e : A \qquad B \hookrightarrow A}{\Gamma \vdash \mathsf{cast}_{\uparrow}\ [B]\ e : B} \qquad \textsc{T-CastDn}\ \frac{\Gamma \vdash e : A \qquad A \hookrightarrow B}{\Gamma \vdash \mathsf{cast}_{\downarrow}\ e : B}$$

$\boxed{\vdash \Gamma}$ *(Well-formedness)*

$$\textsc{W-Nil}\ \frac{}{\vdash \varnothing} \qquad \textsc{W-Cons}\ \frac{\Gamma \vdash A : s \qquad x \text{ fresh in } \Gamma}{\vdash \Gamma, x : A}$$

Figure 4.3. Typing rules of call-by-name PITS

typing, as there is no distinction of different syntactic levels. Most typing rules are quite standard. We write $\vdash \Gamma$ if a context Γ is well-formed. We use $\Gamma \vdash A : s$ to check if A is a well-formed type.

PITS is a *family* of type systems similarly to PTS, parametrized by the axiom set $\mathcal{A} \subseteq (\mathcal{S} \times \mathcal{S})$ for typing sorts and rule set $\mathcal{R} \subseteq (\mathcal{S} \times \mathcal{S} \times \mathcal{S})$ for checking well-formedness of Π-types (see also Section 2.1.1). 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 if $(s_1, s_2, s_3) \in \mathcal{R}$. 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.

Note that unlike the traditional presentation of PTS (see Figure 2.1), rule T-Abs does not rely on rule T-Prod to check the well-formedness of Pi-types, but directly embeds the premises from rule T-Prod. Several PTS-based calculi in the literature such as PTS$_f$ also use such a definition

that can reduce dependencies between rules and simplify proofs of metatheory.

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. 4.2). For example, given a judgment $\Gamma \vdash e : A_2$ and relation $A_1 \hookrightarrow A_2 \hookrightarrow A_3$, $\mathsf{cast}_{\uparrow}\ [A_1]\ e$ expands the type of e from A_2 to A_1, while $\mathsf{cast}_{\downarrow}\ e$ reduces the type of e from A_2 to A_3. We can formally give the typing derivations of the examples in Section 3.1.3:

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

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).

4.1.4 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 μ-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 μ-term itself, resulting in the unfolding $e[x \mapsto \mu x : A.\ e]$. The μ-term is equivalent to a recursive function that should be allowed to unfold without restriction.

Recall the factorial function example in Section 3.1.4. By rule T-MU, the type of $fact$ is $Int \to Int$. Thus we can apply $fact$ to an integer. Note that by rule R-MU, $fact$ will be unfolded to a λ-term. Assuming the evaluation of **if-then-else** construct and arithmetic expressions follows the one-step reduction, we can evaluate the term $fact\ 3$ as follows:

$$\begin{array}{lll} & fact\ 3 & \\ \hookrightarrow & (\lambda x : Int.\ \mathbf{if}\ x == 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times fact\ (x-1))\ 3 & \text{-- by rule R-APP} \\ \hookrightarrow & \mathbf{if}\ 3 == 0\ \mathbf{then}\ 1\ \mathbf{else}\ 3 \times fact\ (3-1) & \text{-- by rule R-BETA} \\ \hookrightarrow & \cdots \hookrightarrow 6 & \end{array}$$

Note that we never check if a μ-term can terminate or not, which is an undecidable problem for general recursive terms. The factorial function example above can stop, while there exist some terms that will loop forever. However, term-level non-termination is only a runtime concern and

does not block the type checker. In Section 4.1.5 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 unfolding (see also Section 2.3.1). In PITS, we do not need to introduce fold and unfold operators, because with the rule R-Mu, $\mathsf{cast}_\uparrow$ and $\mathsf{cast}_\downarrow$ *generalize* fold and unfold, respectively. Suppose that we have terms e_1 and e_2 such that $e_1 : \mu x : A.\ B$ and $e_2 : B[x \mapsto \mu x : A.\ B]$. The type of e_2 is the unfolding of e_1's type, which follows the one-step reduction relation by rule R-Mu:

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

By applying rule T-CastUp and rule T-Castdn, we can obtain the following typing results:

$$\begin{array}{ll} \mathsf{cast}_\downarrow\ e_1 & : B[x \mapsto \mu x : A.\ B] \\ \mathsf{cast}_\uparrow\ [\mu x : A.\ B]\ e_2 & : (\mu x : A.\ B) \end{array}$$

Thus, $\mathsf{cast}_\uparrow$ and $\mathsf{cast}_\downarrow$ witness the isomorphism between the original recursive type and its unfolding, behaving in the same way as fold and unfold in iso-recursive types as in Section 2.3.1.

$$\mu x : A.\ B \underset{\mathsf{cast}_\uparrow\ [\mu x : A.\ B]}{\overset{\mathsf{cast}_\downarrow}{\rightleftharpoons}} B[x \mapsto \mu x : A.\ B]$$

An important remark is that casts are necessary, not only for controlling the unfolding of recursive types, but also for type conversion of other constructs, which is essential for encoding parametrized algebraic datatypes (see Section 3.2.1).

4.1.5 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 and Herbelin 2012] of PITS is decidable without requiring strong normalization. Secondly, *any* PITS is type-safe, proven by subject reduction and progress lemmas [Wright and Felleisen 1994].

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

Definition 4.1.1. *A PITS is functional if:*

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

Such definition is the same as the one of functional PTS [Siles and Herbelin 2012], or *singly-sorted* PTS [Barendregt 1992] (see also Section 2.1.1). Functional PITS enjoys *uniqueness of typing*, i.e., typing result is unique up to alpha equality:

Lemma 4.1.1 (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-CastUp 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 $A \hookrightarrow B$ 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 4.1.2 (Determinacy of One-step Call-by-name Reduction)**.** *If $e \hookrightarrow e_1$ and $e \hookrightarrow e_2$, 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 $\mu x : A.\ e \hookrightarrow e'$. With this result, we show that it is decidable to check whether the one-step relation $A \hookrightarrow B$ holds. We first reduce A by one step to obtain A' (which is unique by Lemma 4.1.2), 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 Π-types are decidable. For other cases, type checking is decidable by the induction hypothesis and uniqueness of typing (see Lemma 4.1.1). Thus, we can conclude the decidability of type checking[3]:

Theorem 4.1.1 (Decidability of Type Checking for Functional PITS)**.** *In any functional PITS, given a well-formed context Γ and a term e, it is decidable to determine if there exists A such that $\Gamma \vdash e : A$.*

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 4.1.2. 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 and Felleisen 1994]:

Theorem 4.1.2 (Subject Reduction of Call-by-name PITS)**.** *If $\Gamma \vdash e : A$ and $e \hookrightarrow e'$ then $\Gamma \vdash e' : A$.*

Theorem 4.1.3 (Progress of Call-by-name PITS)**.** *If $\varnothing \vdash e : A$ then either e is a value v or there exists e' such that $e \hookrightarrow e'$.*

[3] This theorem is also called *decidability of typability* [Barendregt 1992, Section 4.4] where the typing result A is not given and constructed by the typing algorithm.

The proof of subject reduction is straightforward by induction on the derivation of $\Gamma \vdash e : A$ and inversion of $e \hookrightarrow e'$. Some cases need supporting lemmas: rule R-CastElim requires Lemma 4.1.2; rule R-Beta and rule R-Mu require the following substitution lemma:

Lemma 4.1.3 (Substitution of Call-by-name PITS)**.** *If* $\Gamma_1, x : B, \Gamma_2 \vdash e_1 : A$ *and* $\Gamma_1 \vdash e_2 : B$*, then* $\Gamma_1, \Gamma_2[x \mapsto e_2] \vdash e_1[x \mapsto e_2] : A[x \mapsto e_2]$.

The proof of progress is also standard by induction on $\varnothing \vdash e : A$. Notice that $\mathsf{cast}_{\uparrow}\ [A]\ e$ is a value, while $\mathsf{cast}_{\downarrow}\ e_1$ is not: by rule R-Castdn, e_1 will be constantly reduced until it becomes a value that could only be in the form $\mathsf{cast}_{\uparrow}\ [A]\ e$ 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 λ-term but not a $\mathsf{cast}_{\uparrow}$-term. Otherwise, suppose e_1 has the form $\mathsf{cast}_{\uparrow}\ [\Pi x : A.\ B]\ e_1'$. By inversion, we have $\varnothing \vdash e_1' : A'$ and $\Pi x : A.\ B \hookrightarrow A'$. But such A' does not exist because $\Pi x : A.\ B$ is a value which is not reducible.

4.2 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 4.1, 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. We address this problem by using a form of value restriction, inspired by previous work [Swamy et al. 2011; Sjöberg et al. 2012].

4.2.1 Value Restriction

Call-by-value reduction ($\hookrightarrow_{\mathsf{v}}$) 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:

$$\text{RV-Beta}\ \frac{}{(\lambda x : A.\ e)\ v \hookrightarrow_{\mathsf{v}} e[x \mapsto v]} \qquad \text{RV-AppL}\ \frac{e_1 \hookrightarrow_{\mathsf{v}} e_1'}{e_1\ e_2 \hookrightarrow_{\mathsf{v}} e_1'\ e_2} \qquad \text{RV-AppR}\ \frac{e \hookrightarrow_{\mathsf{v}} e'}{v\ e \hookrightarrow_{\mathsf{v}} v\ e'}$$

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:

$$f : (\Pi n : \mathit{Int}.\mathit{Vec}\ n)$$

By rule RV-AppR, the reduction $f\ (1+1) \hookrightarrow_{\mathsf{v}} f\ 2$ holds. However, the types of two sides of the reduction are different:

$$\begin{array}{ll} f\ (1+1) & : \mathit{Vec}\ (1+1) \\ f\ 2 & : \mathit{Vec}\ 2 \end{array}$$

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.

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

$$\text{TV-AppV}\ \frac{\Gamma \vdash_{\mathsf{v}} e : \Pi x : A.\ B \qquad \Gamma \vdash_{\mathsf{v}} v : A}{\Gamma \vdash_{\mathsf{v}} e\ v : B[x \mapsto v]} \qquad \text{TV-App}\ \frac{\Gamma \vdash_{\mathsf{v}} e_1 : A \to B \qquad \Gamma \vdash_{\mathsf{v}} e_2 : A}{\Gamma \vdash_{\mathsf{v}} e_1\ e_2 : B}$$

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 Π-type (see Figure 4.1). Note that these two typing rules overlap: the Π-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 4.2.3). 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 $f\ (1+1) \hookrightarrow_{\mathsf{v}} f\ 2$ 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 become 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., $id\ (x+y)$ where $id = \lambda z : Int.\ z$. In other words, there is no loss of expressiveness with respect to non-dependently typed programming.

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 $Vec\ (1+1) \hookrightarrow_{\mathsf{v}} Vec\ 2$, we can obtain

$$\begin{array}{ll} f\ 2 & : Vec\ 2 \\ \mathsf{cast}_{\uparrow}[Vec\ (1+1)](f\ 2) & : Vec\ (1+1) \end{array}$$

Then, $f\ (1+1) \hookrightarrow_{\mathsf{v}} \mathsf{cast}_{\uparrow}[Vec\ (1+1)](f\ 2)$ preserves the type with an extra $\mathsf{cast}_{\uparrow}$. However, such reduction relation involves types due to the annotation of casts. For simplicity reasons, we

leave this extension as future work (see Section 8.2) and stick to value restriction for a simpler meta-theory.

4.2.2 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., $(\lambda x : Int.\ x)\ y$, where y is a free variable. Using the rule 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 Fireball Calculus [Paolini and Della Rocca 1999], as well as several *open call-by-value* calculi [Accattoli and Guerrieri 2016], 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 unfolding has the form $e[x \mapsto \mu x : A.\ e]$ such that the substituted term is the term itself. Thus, we need to treat *μ-terms as values* as in traditional call-by-value settings. One consequence is that a μ-term now is only unfolded when it is placed at the function part of an application, or inside $\mathsf{cast}_{\downarrow}$:

RV-Mu

$$(\mu x : A.\ e)\ v \hookrightarrow_{\mathsf{v}} (e[x \mapsto \mu x : A.\ e])\ v$$

RV-Castdn-Mu

$$\mathsf{cast}_{\downarrow}\ (\mu x : A.\ e) \hookrightarrow_{\mathsf{v}} \mathsf{cast}_{\downarrow}\ (e[x \mapsto \mu x : A.\ e])$$

Thus, μ-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 μ-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 Variant	*Recursive Type*	*Reduction*
Call-by-name	$f = \mu y : A.\ e$	$f \hookrightarrow e[y \mapsto f]$
Call-by-value	$f' = \mu x : Unit \to A.\ e[y \mapsto x\ ()]$	$f'\ () \hookrightarrow_{\mathsf{v}} (e[y \mapsto x\ ()][x \mapsto f'])\ ()$

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 Figure 4.4. The changes from the call-by-name variant are highlighted.

4.2.3 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:

(Syntax)

Expressions	e, A, B	$::=$	$x \mid s \mid e_1\ e_2 \mid \lambda x : A.\ e \mid \Pi x : A.\ B \mid \mu x : A.\ e$
		$\mid$	$\mathsf{cast}_\uparrow\ [A]\ e \mid \mathsf{cast}_\downarrow\ e$
Values	v	$::=$	$x \mid s \mid \lambda x : A.\ e \mid \Pi x : A.\ B \mid \mu x : A.\ e \mid \mathsf{cast}_\uparrow\ [A]\ v$
Contexts	Γ	$::=$	$\varnothing \mid \Gamma, x : A$

$\boxed{e_1 \hookrightarrow_\mathsf{v} e_2}$ *(Call-by-value Reduction)*

RV-Beta
$$\frac{}{(\lambda x : A.\ e)\ v \hookrightarrow_\mathsf{v} e[x \mapsto v]}$$

RV-Mu
$$\frac{}{(\mu x : A.\ e)\ v \hookrightarrow_\mathsf{v} (e[x \mapsto \mu x : A.\ e])\ v}$$

RV-AppL
$$\frac{e_1 \hookrightarrow_\mathsf{v} e_1'}{e_1\ e_2 \hookrightarrow_\mathsf{v} e_1'\ e_2}$$

RV-AppR
$$\frac{e \hookrightarrow_\mathsf{v} e'}{v\ e \hookrightarrow_\mathsf{v} v\ e'}$$

RV-Castup
$$\frac{e \hookrightarrow_\mathsf{v} e'}{\mathsf{cast}_\uparrow\ [A]\ e \hookrightarrow_\mathsf{v} \mathsf{cast}_\uparrow\ [A]\ e'}$$

RV-Castdn
$$\frac{e \hookrightarrow_\mathsf{v} e'}{\mathsf{cast}_\downarrow\ e \hookrightarrow_\mathsf{v} \mathsf{cast}_\downarrow\ e'}$$

RV-Castdn-Mu
$$\frac{}{\mathsf{cast}_\downarrow\ (\ \mu x : A.\ e\ )\ \hookrightarrow_\mathsf{v} \mathsf{cast}_\downarrow\ (\ e[x \mapsto \mu x : A.\ e]\ )}$$

RV-CastElim
$$\frac{}{\mathsf{cast}_\downarrow\ (\mathsf{cast}_\uparrow\ [A]\ v)\ \hookrightarrow_\mathsf{v} v}$$

$\boxed{\Gamma \vdash_\mathsf{v} e : A}$ *(Typing of Call-by-value PITS)*

TV-Ax
$$\frac{\vdash_\mathsf{v} \Gamma \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash_\mathsf{v} s_1 : s_2}$$

TV-Var
$$\frac{\vdash_\mathsf{v} \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash_\mathsf{v} x : A}$$

TV-Abs
$$\frac{\Gamma \vdash_\mathsf{v} A : s_1 \qquad \Gamma, x : A \vdash_\mathsf{v} e : B \qquad \Gamma, x : A \vdash_\mathsf{v} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_\mathsf{v} \lambda x : A.\ e : \Pi x : A.\ B}$$

TV-App
$$\frac{\Gamma \vdash_\mathsf{v} e_1 : A \to B \qquad \Gamma \vdash_\mathsf{v} e_2 : A}{\Gamma \vdash_\mathsf{v} e_1\ e_2 : B}$$

TV-AppV
$$\frac{\Gamma \vdash_\mathsf{v} e : \Pi x : A.\ B \qquad \Gamma \vdash_\mathsf{v} v : A}{\Gamma \vdash_\mathsf{v} e\ v : B[x \mapsto v]}$$

TV-Prod
$$\frac{\Gamma \vdash_\mathsf{v} A : s_1 \qquad \Gamma, x : A \vdash_\mathsf{v} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_\mathsf{v} \Pi x : A.\ B : s_3}$$

TV-Mu
$$\frac{\Gamma \vdash_\mathsf{v} A : s \qquad \Gamma, x : A \vdash_\mathsf{v} e : A}{\Gamma \vdash_\mathsf{v} \mu x : A.\ e : A}$$

TV-Castup
$$\frac{\Gamma \vdash_\mathsf{v} B : s \qquad \Gamma \vdash_\mathsf{v} e : A \qquad B \hookrightarrow_\mathsf{v} A}{\Gamma \vdash_\mathsf{v} \mathsf{cast}_\uparrow\ [B]\ e : B}$$

TV-Castdn
$$\frac{\Gamma \vdash_\mathsf{v} e : A \qquad A \hookrightarrow_\mathsf{v} B}{\Gamma \vdash_\mathsf{v} \mathsf{cast}_\downarrow\ e : B}$$

$\boxed{\vdash_\mathsf{v} \Gamma}$ *(Well-formedness)*

WV-Nil
$$\frac{}{\vdash_\mathsf{v} \varnothing}$$

WV-Cons
$$\frac{\Gamma \vdash_\mathsf{v} A : s \qquad x \text{ fresh in } \Gamma}{\vdash_\mathsf{v} \Gamma, x : A}$$

Figure 4.4. Call-by-value PITS

Lemma 4.2.1 (Substitution of Call-by-value PITS). *If* $\Gamma_1, x : B, \Gamma_2 \vdash_\mathsf{v} e : A$ *and* $\Gamma_1 \vdash_\mathsf{v} v : B$, *then* $\Gamma_1, \Gamma_2[x \mapsto v] \vdash_\mathsf{v} e[x \mapsto v] : A[x \mapsto v]$.

We now require substituted terms to be values. With such a change, type preservation of reducing open terms is possible, as discussed in Section 4.2.2. Such restricted substitution lemma is sufficient

for proving subject reduction, because all substituted terms are values in reduction rules (see Figure 4.4). The subject reduction and progress lemmas can be proved in a similar way to call-by-name PITS:

Theorem 4.2.1 (Subject Reduction of Call-by-value PITS). *If* $\Gamma \vdash_{\mathsf{v}} e : A$ *and* $e \hookrightarrow_{\mathsf{v}} e'$ *then* $\Gamma \vdash_{\mathsf{v}} e' : A$.

Theorem 4.2.2 (Progress of Call-by-value PITS). *If* $\varnothing \vdash_{\mathsf{v}} e : A$ *then either* e *is a value* v *or there exists* e' *such that* $e \hookrightarrow_{\mathsf{v}} e'$.

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

Lemma 4.2.2 (Determinacy of One-step Call-by-value Reduction). *If* $e \hookrightarrow_{\mathsf{v}} e_1$ *and* $e \hookrightarrow_{\mathsf{v}} e_2$, *then* $e_1 \equiv e_2$.

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

Lemma 4.2.3 (Uniqueness of Typing for Functional PITS). *In any functional PITS, if* $\Gamma \vdash_{\mathsf{v}} e : A$ *and* $\Gamma \vdash_{\mathsf{v}} e : B$, *then* $A \equiv B$.

Theorem 4.2.3 (Decidability of Type Checking for Functional PITS). *In any functional PITS, given a well-formed context* Γ *and a term* e, *it is decidable to determine if there exists* A *such that* $\Gamma \vdash_{\mathsf{v}} e : A$.

4.3 Iso-Types with Full Casts

In Sections 4.1 and 4.2, 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 (e.g. cannot encode the vector example in Section 3.2.4). 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*. The proofs and metatheory design is inspired by approaches used in Zombie [Sjöberg and Weirich 2015] and Guru [Stump et al. 2008].

4.3.1 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 $Vec\ (1 + 1)$ to $Vec\ 2$, 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 ($\mathsf{cast}_{\Uparrow}$ and $\mathsf{cast}_{\Downarrow}$) utilizing a *one-step decidable parallel reduction* ($\hookrightarrow_{\mathsf{p}}$) relation for type conversion. Figure 4.5 shows the definition of $\hookrightarrow_{\mathsf{p}}$. This relation allows reducing terms at any position, including non-head positions or inside binders, e.g., $\lambda x : Int.\ 1 + 1 \hookrightarrow_{\mathsf{p}} \lambda x : Int.\ 2$. Thus full type-level computation for casts is enabled.

$\boxed{r_1 \hookrightarrow_{\mathsf{p}} r_2}$ *(Decidable Parallel Reduction)*

$$\textsc{P-Red}\ \frac{}{(\lambda x : R.\ r_1)\ r_2 \hookrightarrow_{\mathsf{p}} r_1[x \mapsto r_2]} \qquad \textsc{P-Var}\ \frac{}{x \hookrightarrow_{\mathsf{p}} x} \qquad \textsc{P-Sort}\ \frac{}{s \hookrightarrow_{\mathsf{p}} s} \qquad \textsc{P-App}\ \frac{r_1 \hookrightarrow_{\mathsf{p}} r_1' \qquad r_2 \hookrightarrow_{\mathsf{p}} r_2'}{r_1\ r_2 \hookrightarrow_{\mathsf{p}} r_1'\ r_2'}$$

$$\textsc{P-Abs}\ \frac{R \hookrightarrow_{\mathsf{p}} R' \qquad r \hookrightarrow_{\mathsf{p}} r'}{\lambda x : R.\ r \hookrightarrow_{\mathsf{p}} \lambda x : R'.\ r'} \qquad \textsc{P-Prod}\ \frac{R_1 \hookrightarrow_{\mathsf{p}} R_1' \qquad R_2 \hookrightarrow_{\mathsf{p}} R_2'}{\Pi x : R_1.\ R_2 \hookrightarrow_{\mathsf{p}} \Pi x : R_1'.\ R_2'} \qquad \textsc{P-MuRed}\ \frac{}{\mu x : R.\ r \hookrightarrow_{\mathsf{p}} r[x \mapsto \mu x : R.\ r]}$$

$$\textsc{P-Mu}\ \frac{R \hookrightarrow_{\mathsf{p}} R' \qquad r \hookrightarrow_{\mathsf{p}} r'}{\mu x : R.\ r \hookrightarrow_{\mathsf{p}} \mu x : R'.\ r'}$$

Figure 4.5. One-step decidable parallel reduction of erased terms

$\boxed{|e|}$ *(Term Erasure)*

$$\begin{array}{lcl}
|x| & = & x \\
|s| & = & s \\
|e_1\ e_2| & = & |e_1|\ |e_2| \\
|\lambda x : A.\ e| & = & \lambda x : |A|.\ |e| \\
|\Pi x : A.\ B| & = & \Pi x : |A|.\ |B| \\
|\mu x : A.\ e| & = & \mu x : |A|.\ |e| \\
|\mathsf{cast}_{\Uparrow}\ [A]\ e| & = & |e| \\
|\mathsf{cast}_{\Downarrow}\ [A]\ e| & = & |e|
\end{array}$$

$\boxed{|\Gamma|}$ *(Context Erasure)*

$$\begin{array}{lcl}
|\varnothing| & = & \varnothing \\
|\Gamma, x : A| & = & |\Gamma|, x : |A|
\end{array}$$

Figure 4.6. Erasure of casts

There are three remarks for parallel reduction worth mentioning. Firstly, parallel reduction is defined up to *erasure* denoted by $|e|$ (see Figure 4.6), 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*. Note that casts in call-by-name PITS do not have this property, as we define $\mathsf{cast}_{\uparrow}\ [A]\ e$ to be a value. If we erase the cast operator, e can be further reducible, which changes the dynamic semantics. In full (call-by-value) PITS, $\mathsf{cast}_{\Uparrow}\ [A]\ v$ ($\mathsf{cast}_{\uparrow}\ [A]\ v$) is a value and becomes v after erasure, which is still a value and does not change the computational behavior.

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 PTS_{μ}. The syntax and semantics of PTS_{μ} is shown in Figure 4.7.

(Syntax)

$$
\begin{array}{lrcl}
\text{Expressions} & r, R & ::= & x \mid s \mid r_1\ r_2 \mid \lambda x : R.\ r \mid \Pi x : R_1.\ R_2 \mid \mu x : R.\ r \\
\text{Values} & u & ::= & s \mid \lambda x : R.\ r \mid \Pi x : R_1.\ R_2 \\
\text{Contexts} & \Delta & ::= & \varnothing \mid \Delta, x : R
\end{array}
$$

$\boxed{r_1 \hookrightarrow r_2}$ *(Weak-head Reduction)*

$$\text{RE-Beta}\ \frac{}{(\lambda x : R.\ r_1)\ r_2 \hookrightarrow r_1[x \mapsto r_2]} \qquad \text{RE-App}\ \frac{r_1 \hookrightarrow r_1'}{r_1\ r_2 \hookrightarrow r_1'\ r_2} \qquad \text{RE-Mu}\ \frac{}{\mu x : R.\ r \hookrightarrow r[x \mapsto \mu x : R.\ r]}$$

$\boxed{\Delta \vdash r : R}$ *(Typing of PTS$_\mu$)*

$$\text{TE-Ax}\ \frac{\vdash \Delta \qquad (s_1, s_2) \in \mathcal{A}}{\Delta \vdash s_1 : s_2} \qquad \text{TE-Var}\ \frac{\vdash \Delta \qquad x : R \in \Delta}{\Delta \vdash x : R}$$

$$\text{TE-Abs}\ \frac{\Delta \vdash R_1 : s_1 \qquad \Delta, x : R_1 \vdash r : R_2 \qquad \Delta, x : R_1 \vdash R_2 : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Delta \vdash \lambda x : R_1.\ r : \Pi x : R_1.\ R_2}$$

$$\text{TE-App}\ \frac{\Delta \vdash r_1 : \Pi x : R_1.\ R_2 \qquad \Delta \vdash r_2 : R_1}{\Delta \vdash r_1\ r_2 : R_2[x \mapsto r_2]} \qquad \text{TE-Prod}\ \frac{\Delta \vdash R_1 : s_1 \qquad \Delta, x : R_1 \vdash R_2 : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Delta \vdash \Pi x : R_1.\ R_2 : s_3}$$

$$\text{TE-Mu}\ \frac{\Delta \vdash R : s \qquad \Delta, x : R \vdash r : R}{\Delta \vdash \mu x : R.\ r : R} \qquad \text{TE-Conv}\ \frac{\Delta \vdash r : R_1 \qquad \Delta \vdash R_2 : s \qquad R_1 \equiv_\beta R_2}{\Delta \vdash r : R_2}$$

$\boxed{\vdash \Delta}$ *(Well-formedness)*

$$\text{WE-Nil}\ \frac{}{\vdash \varnothing} \qquad \text{WE-Cons}\ \frac{\Delta \vdash R : s \qquad x \text{ fresh in } \Gamma}{\vdash \Delta, x : R}$$

Figure 4.7. PTS$_\mu$

Secondly, the definition of parallel reduction in Figure 4.5 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 unfolding, respectively. The confluence property for one-step reduction is lost[4]. Nevertheless, such definition makes the decidability property (see Lemma 4.3.5) 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 4.1.2). For example, for the term $(\lambda x : Int.\ 1 + 1)\ 3$, we can (parallel) reduce it to either $(\lambda x : Int.\ 2)\ 3$ 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 $\mathsf{cast}_\Downarrow$ operator to indicate what exact type we want to reduce to. Similarly to $\mathsf{cast}_\Uparrow$, $\mathsf{cast}_\Downarrow\ [A]\ v$ is a value, which is different from the

[4]Notice that multi-step reduction $\twoheadhookrightarrow_{\mathsf{p}}$ is still confluent since it is equivalent to multi-step full beta reduction $\longrightarrow_\beta^*$ (see Lemma 4.3.3) which is confluent.

(Syntax)

$$
\begin{array}{llll}
\text{Expressions} & e, A, B & ::= & x \mid s \mid e_1\ e_2 \mid \lambda x : A.\ e \mid \Pi x : A.\ B \\
 & & \mid & \mu x : A.\ e \mid \mathsf{cast}_{\Uparrow}\ [A]\ e \mid \mathsf{cast}_{\Downarrow}\ [A]\ e \\
\text{Contexts} & \Gamma & ::= & \varnothing \mid \Gamma, x : A
\end{array}
$$

$\boxed{\Gamma \vdash_{\mathsf{f}} e : A}$ *(Typing of Full PITS)*

$$\textsc{TF-Ax}\ \frac{\vdash_{\mathsf{f}} \Gamma \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash_{\mathsf{f}} s_1 : s_2} \qquad \textsc{TF-Var}\ \frac{\vdash_{\mathsf{f}} \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash_{\mathsf{f}} x : A}$$

$$\textsc{TF-Abs}\ \frac{\Gamma \vdash_{\mathsf{f}} A : s_1 \qquad \Gamma, x : A \vdash_{\mathsf{f}} e : B \qquad \Gamma, x : A \vdash_{\mathsf{f}} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_{\mathsf{f}} \lambda x : A.\ e : \Pi x : A.\ B}$$

$$\textsc{TF-App}\ \frac{\Gamma \vdash_{\mathsf{f}} e_1 : \Pi x : A.\ B \qquad \Gamma \vdash_{\mathsf{f}} e_2 : A}{\Gamma \vdash_{\mathsf{f}} e_1\ e_2 : B[x \mapsto e_2]} \qquad \textsc{TF-Prod}\ \frac{\Gamma \vdash_{\mathsf{f}} A : s_1 \qquad \Gamma, x : A \vdash_{\mathsf{f}} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_{\mathsf{f}} \Pi x : A.\ B : s_3}$$

$$\textsc{TF-Mu}\ \frac{\Gamma \vdash_{\mathsf{f}} A : s \qquad \Gamma, x : A \vdash_{\mathsf{f}} e : A}{\Gamma \vdash_{\mathsf{f}} \mu x : A.\ e : A} \qquad \textsc{TF-Castup}\ \frac{\Gamma \vdash_{\mathsf{f}} B : s \qquad \Gamma \vdash_{\mathsf{f}} e : A \qquad |B| \hookrightarrow_{\mathsf{p}} |A|}{\Gamma \vdash_{\mathsf{f}} \mathsf{cast}_{\Uparrow}\ [B]\ e : B}$$

$$\textsc{TF-Castdn}\ \frac{\Gamma \vdash_{\mathsf{f}} B : s \qquad \Gamma \vdash_{\mathsf{f}} e : A \qquad |A| \hookrightarrow_{\mathsf{p}} |B|}{\Gamma \vdash_{\mathsf{f}} \mathsf{cast}_{\Downarrow}\ [B]\ e : B}$$

$\boxed{\vdash_{\mathsf{f}} \Gamma}$ *(Well-formedness)*

$$\textsc{WF-Nil}\ \frac{}{\vdash_{\mathsf{f}} \varnothing} \qquad \textsc{WF-Cons}\ \frac{\Gamma \vdash_{\mathsf{f}} A : s \qquad x \text{ fresh in } \Gamma}{\vdash_{\mathsf{f}} \Gamma, x : A}$$

Figure 4.8. Full PITS

call-by-name/value variant.

Figure 4.8 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_μ 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_μ (see Figure 4.7). 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 ($\hookrightarrow$) for term evaluation for erased terms in PTS_μ (see Figure 4.7).

4.3.2 Metatheory

We show that the two key properties, type safety and decidability of type checking, still hold in full PITS.

$\boxed{r_1 \longrightarrow_\beta r_2}$ *(Full Reduction)*

$$\textsc{B-Red}\ \frac{}{(\lambda x : R.\ r_1)\ r_2 \longrightarrow_\beta r_1[x \mapsto r_2]} \qquad \textsc{B-App1}\ \frac{r_1 \longrightarrow_\beta r_1'}{r_1\ r_2 \longrightarrow_\beta r_1'\ r_2} \qquad \textsc{B-App2}\ \frac{r_2 \longrightarrow_\beta r_2'}{r_1\ r_2 \longrightarrow_\beta r_1\ r_2'}$$

$$\textsc{B-Abs1}\ \frac{R \longrightarrow_\beta R'}{\lambda x : R.\ r \longrightarrow_\beta \lambda x : R'.\ r} \qquad \textsc{B-Abs2}\ \frac{r \longrightarrow_\beta r'}{\lambda x : R.\ r \longrightarrow_\beta \lambda x : R.\ r'} \qquad \textsc{B-Prod1}\ \frac{R_1 \longrightarrow_\beta R_1'}{\Pi x : R_1.\ R_2 \longrightarrow_\beta \Pi x : R_1'.\ R_2}$$

$$\textsc{B-Prod2}\ \frac{R_2 \longrightarrow_\beta R_2'}{\Pi x : R_1.\ R_2 \longrightarrow_\beta \Pi x : R_1.\ R_2'} \qquad \textsc{B-MuRed}\ \frac{}{\mu x : R.\ r \longrightarrow_\beta r[x \mapsto \mu x : R.\ r]} \qquad \textsc{B-Mu1}\ \frac{R \longrightarrow_\beta R'}{\mu x : R.\ r \longrightarrow_\beta \mu x : R'.\ r}$$

$$\textsc{B-Mu2}\ \frac{r \longrightarrow_\beta r'}{\mu x : R.\ r \longrightarrow_\beta \mu x : R.\ r'}$$

Figure 4.9. Full beta reduction

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,

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

cannot be further reduced because the head position is already a value but not a λ-term. Note that weak casts do not have such problem because only $\mathsf{cast}_{\uparrow}$ is annotated and it is not legal to have a Π-type in the annotation (see last paragraph of Section 4.1.5). 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 pushing $\mathsf{cast}_{\Downarrow}$ into the λ-term:

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

However, adding "push rules" significantly complicates the reduction relations and metatheory. Instead, we adopt the erasure approach inspired by Zombie [Sjöberg and Weirich 2015] 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_μ (see Figure 4.7). The typing judgment is $\Delta \vdash r : R$ where Δ ranges over the erased context.

PTS_μ is a variant of PTS with recursion. We follow the standard proof steps for PTS [Barendregt 1992]. The substitution and progress lemmas are stated as follows:

Lemma 4.3.1 (Substitution of PTS_μ)**.** *If $\Delta_1, x : R', \Delta_2 \vdash r_1 : R$ and $\Delta_1 \vdash r_2 : R'$, then $\Delta_1, \Delta_2[x \mapsto r_2] \vdash r_1[x \mapsto r_2] : R[x \mapsto r_2]$.*

Theorem 4.3.1 (Progress of PTS_μ)**.** *If $\varnothing \vdash r : R$ then either r is a value u or there exists r' such that $r \hookrightarrow r'$.*

Notice that term evaluation uses the weak-head reduction $\hookrightarrow$. We only need to prove subject reduction and progress theorems for $\hookrightarrow$. But we generalize the result for subject reduction, which holds up to the parallel reduction $\hookrightarrow_{\mathsf{p}}$. We first show subject reduction holds for one-step full

beta reduction $\longrightarrow_\beta$ (see Figure 4.9) and multi-step full beta reduction $\longrightarrow_\beta^*$, i.e., reflexive and transitive closure of $\longrightarrow_\beta$:

Lemma 4.3.2 (Subject Reduction for Full Beta Reduction). *1. If $\Delta \vdash r : R$ and $r \longrightarrow_\beta r'$ then $\Delta \vdash r' : R$.*

2. If $\Delta \vdash r : R$ and $r \longrightarrow_\beta^* r'$ then $\Delta \vdash r' : R$.

Then we show $\longrightarrow_\beta^*$ is equivalent to multi-step parallel reduction $\twoheadrightarrow_{\mathsf{p}}$, i.e., transitive closure of $\hookrightarrow_{\mathsf{p}}$ (since it is already reflexive):

Lemma 4.3.3 (Equivalence of Parallel Reduction). *Given r_1 and r_2, $r_1 \longrightarrow_\beta^* r_2$ holds if and only if $r_1 \twoheadrightarrow_{\mathsf{p}} r_2$ holds.*

Thus, subject reduction for parallel reduction $\hookrightarrow_{\mathsf{p}}$ is an immediate corollary:

Theorem 4.3.2 (Subject Reduction for Parallel Reduction). *If $\Delta \vdash r : R$ and $r \hookrightarrow_{\mathsf{p}} r'$ then $\Delta \vdash r' : R$.*

Finally, given that the PTS$_\mu$ 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 4.3.4 (Soundness of Erasure). *If $\Gamma \vdash_{\mathsf{f}} e : A$ 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 4.1.5. We also limit discussion of decidability to functional PITS (see Definition 4.1.1). The only difference is for cast rule TF-CastUp and rule TF-CastDn, which use parallel reduction $|A_1| \hookrightarrow_{\mathsf{p}} |A_2|$ as a premise. We first show the decidability of parallel reduction:

Lemma 4.3.5 (Decidability of Parallel Reduction). *Given r_1 and r_2, it is decidable to determine whether $r_1 \hookrightarrow_{\mathsf{p}} r_2$ holds.*

The proof is by induction on the length of r_1 and does not rely on the single-step confluence of $\hookrightarrow_{\mathsf{p}}$. The confluence of $\hookrightarrow_{\mathsf{p}}$ 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 unfolding, respectively (see Section 4.3.1). However, both rules become *deterministic*, which makes it easier to determine if $\hookrightarrow_{\mathsf{p}}$ 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 4.1.5). 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., $\lambda x : R_3.\ r_3 \hookrightarrow_{\mathsf{p}} r_5$ and $r_4 \hookrightarrow_{\mathsf{p}} r_6$ hold, then $r_1 \hookrightarrow_{\mathsf{p}} r_2$ holds trivially. Otherwise, $r_1 \hookrightarrow_{\mathsf{p}} r_2$ holds if and only if P-Red holds. The latter can be determined by testing the equality of $r_3[x \mapsto r_4]$ 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

$$\frac{r_1 \hookrightarrow_{\mathsf{p}} r_1' \qquad r_2 \hookrightarrow_{\mathsf{p}} r_2'}{(\lambda x : R.\ r_1)\ r_2 \hookrightarrow_{\mathsf{p}} r_1'[x \mapsto r_2']}$$

We now need to determine if $r_3 \hookrightarrow_{\mathsf{p}} r_3'$ holds or not. This is non-trivial since the induction hypothesis only gives the hint whether $\lambda x : R_3.\ r_3 \hookrightarrow_{\mathsf{p}} r_5$ holds but not directly for r_3.

$\boxed{\Delta \vDash r : R}$ *(Typing of PTS$_{step}$)*

$$\textsc{TS-Ax}\ \frac{\vDash \Delta \qquad (s_1, s_2) \in \mathcal{A}}{\Delta \vDash s_1 : s_2} \qquad \textsc{TS-Var}\ \frac{\vDash \Delta \qquad x : R \in \Delta}{\Delta \vDash x : R}$$

$$\textsc{TS-Abs}\ \frac{\Delta \vDash R_1 : s_1 \qquad \Delta, x : R_1 \vDash r : R_2 \qquad \Delta, x : R_1 \vDash R_2 : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Delta \vDash \lambda x : R_1.\ r : \Pi x : R_1.\ R_2}$$

$$\textsc{TS-App}\ \frac{\Delta \vDash r_1 : \Pi x : R_1.\ R_2 \qquad \Delta \vDash r_2 : R_1}{\Delta \vDash r_1\ r_2 : R_2[x \mapsto r_2]} \qquad \textsc{TS-Prod}\ \frac{\Delta \vDash R_1 : s_1 \qquad \Delta, x : R_1 \vDash R_2 : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Delta \vDash \Pi x : R_1.\ R_2 : s_3}$$

$$\textsc{TS-mu}\ \frac{\Delta \vDash R : s \qquad \Delta, x : R \vDash r : R}{\Delta \vDash \mu x : R.\ r : R} \qquad \textsc{TS-Betaup}\ \frac{\Delta \vDash r : R_1 \qquad \Delta \vDash R_2 : s \qquad R_2 \longrightarrow_\beta R_1}{\Delta \vDash r : R_2}$$

$$\textsc{TS-Betadn}\ \frac{\Delta \vDash r : R_1 \qquad \Delta \vDash R_2 : s \qquad R_1 \longrightarrow_\beta R_2}{\Delta \vDash r : R_2}$$

$\boxed{\vDash \Delta}$ *(Well-formedness)*

$$\textsc{WS-Nil}\ \frac{}{\vDash \varnothing} \qquad \textsc{WS-Cons}\ \frac{\Delta \vDash R : s \qquad x \text{ fresh in } \Gamma}{\vDash \Delta, x : R}$$

Figure 4.10. Typing rules of PTS$_{\text{step}}$

As $\mathsf{cast}_{\Uparrow}$ and $\mathsf{cast}_{\Downarrow}$ are annotated, both A_1 and A_2 can be determined and the well-typedness is checked in the original system. By Lemma 4.3.4, the erased terms keep the well-typedness. By Lemma 4.3.5, it is decidable to check if $|A_1| \hookrightarrow_{\mathsf{p}} |A_2|$. We conclude the decidability of type checking by following lemmas:

Lemma 4.3.6 (Uniqueness of Typing for functional PITS)**.** *In any functional PITS, if* $\Gamma \vdash_{\mathsf{f}} e : A_1$ *and* $\Gamma \vdash_{\mathsf{f}} e : A_2$*, then* $A_1 \equiv A_2$*.*

Theorem 4.3.3 (Decidability of Type Checking for functional PITS)**.** *In any functional PITS, given a well-formed context* Γ *and a term* e*, it is decidable to determine if there exists* A *such that* $\Gamma \vdash_{\mathsf{f}} e : A$*.*

4.3.3 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]:

$$\textsc{TS-Betaup}\ \frac{\Delta \vDash r : R_1 \qquad \Delta \vDash R_2 : s \qquad R_2 \longrightarrow_\beta R_1}{\Delta \vDash r : R_2} \qquad \textsc{TS-Betadn}\ \frac{\Delta \vDash r : R_1 \qquad \Delta \vDash R_2 : s \qquad R_1 \longrightarrow_\beta R_2}{\Delta \vDash r : R_2}$$

where $\longrightarrow_\beta$ denotes full beta reduction (see Figure 4.9). We call this variant PTS_{step} (see Figure 4.10). Its typing judgment is denoted by $\Delta \vDash r : R$. PITS can be seen as an annotated version of PTS_{step}. We first show that full PITS is complete to PTS_{step}:

Lemma 4.3.7 (Completeness of Full PITS to PTS_{step})**.** *If $\Delta \vDash r : R$, then there exists Γ, e and A such that $\Gamma \vdash_{\mathsf{f}} e : A$ 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 and Herbelin 2012]). We have the following relation between PTS_{step} and PTS_μ:

Lemma 4.3.8 (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_μ:

Theorem 4.3.4 (Completeness of Full PITS to PTS_μ)**.** *If $\Delta \vdash r : R$, then there exists Γ, e, A such that $\Gamma \vdash_{\mathsf{f}} e : A$ where $|\Gamma| = \Delta$, $|e| = r$ and $|A| = R$.*

4.4 Discussion

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_f) [van Doorn et al. 2013], which is a variant of PTS that replaces the conversion rule by *embedding* explicit conversion steps into terms. PTS_f has strong connections to PITS in the sense that both systems are based on Pure Type Systems and require *explicit* type conversions — PTS_f uses proof-annotated terms, while PITS uses cast operators. Although PTS_f was motivated by applications to theorem proving, PTS_f (like PTS) can be instantiated to form inconsistent calculi, which can encode fixpoints and general recursion. Therefore PTS_f could also, in principle, be used as a foundation for programming languages. However, the underlying mechanisms and foundations of PTS_f 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 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. In the rest of this section, we give a detailed comparison on features between PTS_f and the three variants of PITS, summarized in Table 4.1. Other closely related work will be discussed in Section 7.1.

4.4.1 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

Table 4.1. Comparison between PTS_f and PITS

Features	PTS_f	Call-by-name PITS	Call-by-value PITS	Full PITS
Direct Dynamic Semantics	○	●	●	○
Direct Proofs	○	●	●	◐[1]
No Mutually Dependent Judgments	○	●	●	●
Implicit Proofs by Reduction	○	●	●	●
Full Type-Level Computation	●	○	○	●
Consistency of Reduction	–	●	●	○
Decidability with Recursion	◐[2]	●	●	●
SLOC of Coq proofs	7318	1217	1477	3796
Lemmas of Coq proofs	319	62	66	221

[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].

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 4.3.2, 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; Sjöberg and 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 4.3.2).

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_f, 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_f 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) (see Section 8.2).

4.4.2 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 4.1 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_μ along with full PITS.

PTS_f 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_f and PTS is highly non-trivial [van Doorn et al. 2013; Siles and Herbelin 2012]. In PTS_f there is no discussion on proving subject reduction *directly* in PTS_f. The type-safety of PTS_f 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 4.3.2). The formalization of PTS_f requires about 7000 SLOC and 300 lemmas (see Table 4.1), including auxiliary systems such as plain PTS and PTS_e. 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_f does not include proofs of decidability of type-checking. Nevertheless the numbers are useful to give an idea of the effort in the PTS_f 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. For example, we will study a non-trivial extension to PITS, i.e. subtyping, in the second part of the thesis. Integrating subtyping and some form of dependent types is a widely acknowledged difficult problem [Aspinall and Compagnoni 1996; Castagna and Chen 2001; Hutchins 2010]. Nevertheless, in Chapter 5 we will show that using the call-by-name instance of PITS extension with subtyping, we can develop a calculus that subsumes System $F_{\leqslant}$ [Cardelli et al. 1994] and has several interesting properties, including subject reduction and transitivity of subtyping. We believe this development would be a lot harder to do by extending full PITS or PTS_f.

4.4.3 No Mutually Dependent Judgments

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. Moreover, 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 ($A \hookrightarrow B$), 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 PTS_f the mutually dependent judgments leads to several lemmas that need to be mutually proved.

4.4.4 Implicit Proofs by Reduction

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 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 PTS_f are more "explicit" than in PITS. One needs to specify which proof terms to be used exactly in 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 ($1 + 1 \hookrightarrow 2$):

$$\begin{aligned} e &: \Pi x : \mathit{Int}.\ \mathit{Vec}\ (1 + 1) \\ e' &: \Pi x : \mathit{Int}.\ \mathit{Vec}\ 2 \end{aligned}$$

To obtain e' from e, in PTS_f, $e' = e^H$ where H is a proof term such that

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

In full PITS, $e' = \mathsf{cast}_{\Downarrow}\ [\Pi x : \mathit{Int}.\ \mathit{Vec}\ 2]\ e$, 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 $\mathsf{cast}_{\downarrow}$ operator even does not require a type annotation. For example, consider a simpler type conversion $(\lambda x : \star.\ x)\ \mathit{Int} \hookrightarrow \mathit{Int}$ from Section 3.1.3:

$$\begin{aligned} e &: (\lambda x : \star.\ x)\ \mathit{Int} \\ e' &: \mathit{Int} \end{aligned}$$

where e' can simply be $\mathsf{cast}_{\downarrow}\ e$ 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 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 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.

4.4.5 Full Type-Level Computation

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 3.2 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 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.

4.4.6 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, term "$(\lambda x : \mathit{Int}.\ 1)\ \bot$" loops in call-by-value reduction, but terminates in call-by-name reduction. 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 run-time semantics/reduction then we need to ensure that the two reductions are in some sense consistent. To (trivially) ensure consistency call-by-name/value PITS uses the same reduction relation for both term evaluation and type conversion. In full PITS, the use of parallel reduction breaks consistency because type-level reduction allows some reductions that are not allowed by term-level reduction. For example, $\mathit{Int} \to \mathit{Vec}\ (1 + 1)$ can be reduced to $\mathit{Int} \to \mathit{Vec}\ 2$ 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 (see Section 8.2).

PTS_f has no discussion on reduction rules for *term evaluation*, since its dynamic semantics is given by elaboration into PTS. Since the focus of 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.

4.4.7 Decidability in the Presence of Recursion

There is no formal discussion or a 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 4.1.1). Uniqueness of typing is used in the decidability proof of functional PITS (see Section 4.1.5), 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.

Part II:

Iso-Types with Subtyping

CHAPTER 5

Unifying Typing and Subtyping

In this chapter, we present $\lambda I_{\leqslant}$, which is a dependently typed generalization of System $F_{\leqslant}$. The motivation is to design a simple yet expressive calculus that combines dependent types and OOP features. Subtyping is one of the key OOP features but the combination of subtyping and dependent types causes mutual dependency of typing and subtyping judgments and circularity in the metatheory. To address these challenges, $\lambda I_{\leqslant}$ employs a novel technique that unifies *typing* and *subtyping*. In $\lambda I_{\leqslant}$ there is only one judgment that is akin to a typed version of subtyping. Both the typing relation, as well as type well-formedness are just special cases of the subtyping relation. Therefore, $\lambda I_{\leqslant}$ takes a significantly different approach compared to previous work. Previous work essentially attempts to fight the entanglement between typing and subtyping. In contrast, what we propose with $\lambda I_{\leqslant}$ is to embrace such tangling, and essentially collapsing the typing and subtyping relations into the same relation. Our approach is also different from the technique of Pure Subtype Systems by Hutchins [2010], which simply eliminates types and typing. $\lambda I_{\leqslant}$ retains types.

The $\lambda I_{\leqslant}$ calculus employs the idea of iso-types in PITS. It follows the PTS-style unified syntax and contains a single unified syntactic sort that accounts for expressions, types and kinds. Iso-types provide a simple form of type casts, and $\lambda I_{\leqslant}$ adopts that idea to address the issues arising from the combination of recursion and dependent types. The novelty over PITS is the support for OOP features such as *higher-order subtyping* [Pierce and Steffen 1997], *bounded quantification* [Cardelli et al. 1994] and *top types*. To illustrate the expressive power of $\lambda I_{\leqslant}$, we show how object encodings relying on higher-order subtyping can be done in $\lambda I_{\leqslant}$. The resulting calculus enjoys several standard and desirable properties, such as *subject reduction*, *transitivity of subtyping*, *narrowing* as well as standard *substitution lemmas*. We also provide an algorithmic version of $\lambda I_{\leqslant}$ based on bi-directional type-checking [Pierce and Turner 2000], which is shown to be sound and complete with respect to the declarative version. Finally we show that $\lambda I_{\leqslant}$ completely subsumes System $F_{\leqslant}$ in expressive power.

5.1 Overview

In this section, we briefly discuss the problem of combining dependent types with subtyping. We informally introduce the key features of $\lambda I_{\leqslant}$ calculus, namely unified subtyping and the support for dependent types by explicit casts. To illustrate the suitability of $\lambda I_{\leqslant}$ to model objects, we adapt the existential object encoding [Bruce et al. 1999; Pierce and Turner 1994] (originally based

on System $F^{\omega}_{\leqslant}$) to $\lambda I_{\leqslant}$. The formal details of $\lambda I_{\leqslant}$ are further discussed in Sections 5.2 and 5.3.

5.1.1 Unified Syntax versus Stratified Syntax

Calculi with high-order subtyping are usually complex. For example, System $F^{\omega}_{\leqslant}$ [Pierce and Steffen 1997] adds higher-order subtyping and bounded quantification to System F_{ω} and is formalized using *stratified syntax*. Because of the separation of syntax, the subtyping relation in System $F^{\omega}_{\leqslant}$ needs to be defined over multiple syntactic forms of abstraction, i.e., abstraction over terms, types and type operators:

Term abstraction	$\lambda x : A.\ e$
Type abstraction	$\lambda X \leqslant A.\ e$
Operator abstraction	$\lambda X \leqslant A.\ B$

This causes duplication and complexity in the metatheory.

It is tempting to adopt the PTS-style *unified syntax* in System $F^{\omega}_{\leqslant}$ to simplify the subtyping relation. Because System F_{ω} (without subtyping) can be modeled with PTS-style unified syntax: it is a special case of PTSs and covered by Barendregt's λ-cube [Barendregt 1992] (see Section 2.1.2). However, there are several difficulties in applying such simplification to a higher-order system with bounded quantification. Recall that there are three different forms of abstraction in System $F^{\omega}_{\leqslant}$. It is hard to unify them because the abstraction can quantify over a variable using two distinct relations, i.e., typing ($x : A$) and subtyping ($X \leqslant A$). To obtain a uniform representation of abstraction, we need to *unify the typing and subtyping relation* in the first place. Moreover, calculi with PTS-style unified syntax usually allow *dependent types*, e.g., the calculus of constructions [Coquand and Huet 1988]. Combining dependent types and subtyping has its own problems, as discussed in the coming subsection.

5.1.2 Challenges in Combining Subtyping with Dependent Types

Mutual Dependency of Typing and Subtyping. Subtyping and dependent types are well-known features of programming languages. Individually, each of them is well-studied. However, combining them in the same system is usually difficult. A major reason is that allowing dependent types makes the typing and subtyping relations *entangled*. The subtyping and typing[1] judgments become mutually dependent. The typing judgment depends on subtyping because of the *subsumption* rule (see also Section 2.4.1):

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leqslant B}{\Gamma \vdash e : B}$$

Subtyping relations are defined over *well-formed* types, which are checked by the typing judgment in a dependently typed system. For example, the subtyping rule for the top type ($\top$), a universal supertype of any well-formed types (i.e. with kind $\star$), is defined as follows:

$$\frac{\Gamma \vdash A : \star}{\Gamma \vdash A \leqslant \top}$$

[1] Some stratified systems [Aspinall and Compagnoni 1996; Castagna and Chen 2001] also have the *kinding* judgment, which is mutually dependent on typing. We uniformly refer to them as typing.

Circularity in the Metatheory. The mutual dependency causes circularity in the metatheory, since one cannot study properties of subtyping independently from typing. For example, $\lambda P_{\leqslant}$ [Aspinall and Compagnoni 1996] is an extension of the second-order dependently typed calculus λP [Barendregt 1992] with subtyping. In $\lambda P_{\leqslant}$, the substitution lemmas for typing and subtyping depend on each other and require a more complicated proof by simultaneously induction on four different judgments, i.e. subtyping, typing, kinding and formation [Aspinall and Compagnoni 1996]:

$$\begin{array}{ll} \Gamma \vdash A \leqslant B & A \text{ is a subtype of } B \text{ in context } \Gamma \\ \Gamma \vdash M : A & \text{term } M \text{ has type } A \text{ in context } \Gamma \\ \Gamma \vdash A : K & \text{type } A \text{ has kind } K \text{ in context } \Gamma \\ \Gamma \vdash K & K \text{ is a kind in context } \Gamma \end{array}$$

Moreover, $\lambda P_{\leqslant}$ contains the following algorithmic transitivity rule:

$$\frac{\Gamma \vdash (\Gamma(\alpha)\, M_1 \cdots M_n)^{\beta_2} \leqslant A \qquad \alpha \text{ bound in } \Gamma,\ A \not\equiv \alpha\, M_1 \cdots M_n}{\Gamma \vdash \alpha\, M_1 \cdots M_n \leqslant A}$$

The first premise requires to normalize the term $\Gamma(\alpha)\, M_1 \cdots M_n$ using β_2-reduction, a beta reduction relation on types. Thus, the transitivity proof depends on β_2-strong normalization. Also, the transitivity property requires types to be well-formed through β_2-conversion, i.e., the subject reduction of β_2-reduction. As a consequence, the proofs of transitivity, strong normalization and subject reduction depend on each other.

Problems of Existing Solutions. There are several existing options to deal with the circularity. One could carefully prove mutually dependent lemmas together by finding a proper decreasing metric of induction, similar to the proof of substitution lemma in $\lambda P_{\leqslant}$. But such method is usually too specific and cannot be generally applied to other systems, e.g., the substitution proof in $\lambda P_{\leqslant}$ does not apply to $\lambda\Pi^{\&}$ [Castagna and Chen 2001].

Another approach is to break the mutual dependency simply by forbidding typing from occurring in the subtyping judgments. The subtyping judgments are defined over *pre-terms*, terms that may not be well-formed. Then one could prove results about subtyping before typing. An obvious limitation is that subtyping rules that must depend on typing are no longer supported, such as the top type rule shown above. Several systems using this method, such as $PTS^{\leqslant}$ [Zwanenburg 1999], drop the support of top types because of such limitation.

5.1.3 Our Solution: Unified Subtyping

We propose a new approach to solve the circularity problem, which also simplifies the syntax. The $\lambda I_{\leqslant}$ calculus features a single relation for both typing and subtyping, namely *unified subtyping*. The relation has the form:

$$\Gamma \vdash e_1 \leqslant e_2 : A$$

It simultaneously contains the subtyping relation, i.e., e_1 is a subtype of e_2, and the typing relation, i.e., e_1 and e_2 have type A. The ordinary typing judgment can be seen as a *special case* of unified subtyping:

$$\Gamma \vdash e : A \triangleq \Gamma \vdash e \leqslant e : A$$

We solve the circularity problem because typing and subtyping cannot be mutually dependent in the first place: they are essentially the same relation. In $\lambda I_{\leqslant}$, subtyping relations can be defined over well-formed terms. Subtyping rules that depend on typing are allowed without causing mutual dependencies. As a result, top types are supported in $\lambda I_{\leqslant}$. Moreover, the metatheory of $\lambda I_{\leqslant}$ is significantly simplified, e.g., there is only one form of substitution lemma to be proved, as discussed in Section 5.3.

Bounded quantification in $\lambda I_{\leqslant}$. $\lambda I_{\leqslant}$ adopts a unified syntax and supports bounded quantification. Because of the unified representation of typing and subtyping, instead of three separate forms of abstraction in System $F^{\omega}_{\leqslant}$, $\lambda I_{\leqslant}$ has a single form of abstraction: $\lambda x \leqslant e_1 : A.\ e_2$. By convention, the ordinary unbounded abstraction can be treated as syntactic sugar of a top-bounded one:

$$\lambda x : A.\ e \triangleq \lambda x \leqslant \top : A.\ e$$

Notice that the top type ($\top$) is generalized to have any kind A instead of $\star$. With unified syntax, $\lambda I_{\leqslant}$ has fewer language constructs than System $F^{\omega}_{\leqslant}$ and a simpler definition of (unified) subtyping relation (see Section 5.2).

5.1.4 Iso-Types: Dependent Types without Strong Normalization

Most traditional dependently typed languages are strongly normalizing (i.e. all programs terminate). Strong normalization plays a fundamental role in the metatheory of those languages. However, nearly all general-purpose programming languages allow non-terminating programs, so depending on strong normalization is a non-starter if we want to model traditional general-purpose languages. The root of the dependency on strong normalization is the *conversion rule* (see Section 3.1.1). For dependently typed languages with subtyping, the conversion rule is usually subsumed by the subsumption rule (see Section 5.1.2), which requires the subtyping relation $\Gamma \vdash A \leqslant B$ to subsume beta-equivalence $A =_{\beta} B$. Thus, the transitivity of subtyping may also depend on strong normalization if its proof requires to first normalize the types [Aspinall and Compagnoni 1996].

An alternative to the conversion rule. Several existing studies [Stump et al. 2008; Sjöberg et al. 2012; Kimmell et al. 2012; Sjöberg and Weirich 2015; Yang et al. 2016] provide a way to combine general recursion with dependent types, while preserving important properties (such as decidability of type-checking). The key idea is to replace the implicit conversion rule with *explicit type casts*. This has the effect that term/type equality becomes weaker: two terms are only equal up to syntactic equality (not beta-equality). To recover type conversion, an explicit cast must be used. The benefit of this design is that it decouples several properties from strong-normalization.

$\lambda I_{\leqslant}$ adopts the *iso-type* approach [Yang et al. 2016] of PITS introduced in Chapters 3 and 4. The cast operators of $\lambda I_{\leqslant}$ employ the weak-head call-by-name reduction relation (see Section 5.2.2), similarly to the one used by call-by-name PITS (see Section 4.1). This makes the type conversion by casts less expressive than what is provided by the implicit conversion rule. Nevertheless, we do not consider such loss of expressiveness problematic. The absence of conversion rule significantly simplifies the metatheory of $\lambda I_{\leqslant}$ because typing and subtyping are up to alpha-equality and strong normalization is not a necessity for proofs. Since our goal is to design a calculus for traditional programming, we do not require the ability to do *full* type-level computation that is

required for dependently typed programming. Call-by-name casts are still expressive enough for our purposes: to model object encodings. Furthermore, there are alternative designs of casts using call-by-value or full reduction rules (see Sections 4.2 and 4.3), but they introduce some extra complications to the metatheory. Alternative approaches are discussed in Section 5.6.

5.1.5 Example: Object Encodings using $\lambda I_{\leqslant}$

We show an example of object encodings in $\lambda I_{\leqslant}$ using the *existential encoding* method [Pierce and Turner 1994; Bruce et al. 1999] originally based on System $F^{\omega}_{\leqslant}$. The example in this section has two notable differences from the previous example shown in Section 3.2.3 for PITS. First, the object encodings in this section do not rely on Scott-encodings of algebraic datatypes. Instead, we directly encode pairs, records and existential types by Church-encoding *weak dependent sums* in $\lambda I_{\leqslant}$. Second, the example in this section supports *generic message passing* that requires subtyping, which was missing in Section 3.2.3.

Encoding Existential Types and Pairs using Weak Sums. Existential types and pairs are special cases of dependent sums. Existential types specialize A to kind $\star$:

$$\exists x.\ B \triangleq \Sigma x : \star.\ B$$

The constructor and destructor of an existential package are simply **pack** and **unpack** operators of dependent sums, respectively. Pairs are non-dependent sums where x is not free in B. The pair type, constructor and destructors can be encoded as follows:

$$\begin{array}{rcll} A \times B & \triangleq & \Sigma x : A.\ B & \text{where } x \notin \mathsf{FV}(B) \\ (e_1, e_2) & \triangleq & \mathbf{pack}\ [e_1, e_2]\ \mathbf{as}\ \Sigma x : A.\ B & \text{where } x \notin \mathsf{FV}(B) \\ \mathbf{fst}\ e & \triangleq & \mathbf{unpack}\ e\ \mathbf{as}\ [x, y]\ \mathbf{in}\ x & \\ \mathbf{snd}\ e & \triangleq & \mathbf{unpack}\ e\ \mathbf{as}\ [x, y]\ \mathbf{in}\ y & \end{array}$$

where in the encoding of constructor, A and B are types of e_1 and e_2, respectively.

Notice that here we only need the *weak* destructor of dependent sums [Schmidt 1994], i.e., the **unpack** operator that requires x and y are not free in the type C of e_2 (see Section 2.2.1). We use the same Church-encodings of weak sums from Section 2.2.1 and show that subtyping and typing rules of weak sums are admissible in $\lambda I_{\leqslant}$. The proof is trivial and available in Appendix A.1. However, it is non-trivial to Church-encode strong dependent sums without restrictions on **unpack** and using only Pi-types [Cardelli 1986b]. In Chapter 6, we will discuss how to support primitive constructs of strong sums with unified subtyping. For examples in this section, weak dependent sums are sufficient for our purpose to encode existential types and non-dependent pairs and yet more expressive than those constructs. Note that **unpack** operator allows unrestricted projection of existential witnesses:

$$\lambda e : (\Sigma x : A.\ B).\ \mathbf{unpack}\ e\ \mathbf{as}\ [x, y]\ \mathbf{in}\ x$$

No such operation is allowed on existential types in System F or $F_{\leqslant}$ [Amin et al. 2016].

Encoding Objects. Now that pairs and existential types can be encoded in $\lambda I_{\leqslant}$, we present the encoding of objects. Note again that records can be encoded with pairs using standard

techniques [Pierce 2002] and that we assume $\lambda I_{\leqslant}$ is extended with integers, pairs, records and existential types in the following text. We use the same example of cell objects as shown in Section 3.2.3. We first present the existential encoding of objects [Pierce and Turner 1994] in $\lambda I_{\leqslant}$:

$$Obj = \lambda I : \star \to \star.\ \exists X.\ X \times (X \to I\ X)$$

The definition of cell objects [Bruce et al. 1999] is as follows:

$$Cell = \lambda X : \star.\ \{get : Int,\ set : Int \to X,\ bump : X\}$$

Similarly, a cell object c can be defined as follows:

$$\begin{array}{l} c = \mathsf{cast}_{\uparrow}[Obj\ Cell]\ \mathbf{pack}\ [\{x : Int\}, (\{x = 0\},\ \lambda s : \{x : Int\}.\ \mathsf{cast}_{\uparrow}\ [Cell\ \{x : Int\}] \\ \qquad\qquad\qquad\qquad\qquad\qquad \{get = s.x,\ set = \lambda n : Int.\ \{x = n\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\ bump = \{x = s.x + 1\}\}\ )] \\ \qquad\qquad \mathbf{as}\ CellT \end{array}$$

We use the **pack** operator to create an existential package. The type $\{x : Int\}$ corresponds to the existential binder X. The pair afterwards corresponds to the body of the existential type. The first component of the pair is the initial hidden state $\{x = 0\}$. The second component is a function containing three methods that are defined in a record and abstracted by the state variable s. The definition of the three methods follows the cell object interface $Cell$. The result type of the package, i.e., $CellT$, is the one-step reduction of $Obj\ Cell$:

$$CellT = \exists X.\ X \times (X \to Cell\ X)$$

As in Section 3.2.3, we also use two $\mathsf{cast}_{\uparrow}$ operators here: one over the **pack** operator and another over the record of methods. Note that the desired type of the object c (i.e. $Obj\ Cell$) is a type-level application, which is different from $CellT$. We use $\mathsf{cast}_{\uparrow}$ to do one-step type expansion for the package: $Obj\ Cell \hookrightarrow CellT$. Similarly, we use another $\mathsf{cast}_{\uparrow}$ operator in the definition of methods to convert the record type into $Cell\ \{x : Int\}$. We use the following syntactic sugar for consecutive $\mathsf{cast}_{\uparrow}$ and **pack**:

$$\mathbf{pack}\ [A, e]\ \mathbf{up}\ B \triangleq \mathsf{cast}_{\uparrow}\ [B]\ (\mathbf{pack}\ [A, e]\ \mathbf{as}\ B')$$

where $B \hookrightarrow B'$, i.e., B' is the one-step reduction of B.

In addition to the previous example in Section 3.2.3, we can further define message passing to the object by the **unpack** operator to open a package. For example, sending message get to the cell object c is denoted by $c \Leftarrow get$, which is syntactic sugar of the generic message function $getM$:

$$\begin{array}{rcl} c \Leftarrow get & \triangleq & getM\ Cell\ c \\ getM & = & \lambda I \leqslant Cell : \star \to \star.\ \lambda o : Obj\ I. \\ & & \quad \mathbf{unpack}\ (\mathsf{cast}_{\downarrow}\ o)\ \mathbf{as}\ [X, (s, m)]\ \mathbf{in}\ (\mathsf{cast}_{\downarrow}\ (m\ s)).get \end{array}$$

$getM$ is parameterized by interface I and object o with such interface, where I can be any *sub-interface* of $Cell$. We first use the $\mathsf{cast}_{\downarrow}$ operator to convert the type of o from $Obj\ I$ to the existential type $\exists X.\ X \times (X \to I\ X)$. Note that we extend the syntax of **unpack** with

$$
\begin{aligned}
&(c \Leftarrow bump) \Leftarrow get \\
&\quad = (\ (\mathsf{cast}_{\uparrow}[Obj\ Cell]\ \mathbf{pack}\ [\{x : Int\}, (\{x = 0\}, f)]\ \mathbf{as}\ CellT) \Leftarrow bump\ ) \Leftarrow get \\
&\quad = (\ \mathbf{unpack}\ (\mathsf{cast}_{\downarrow}\ (\mathsf{cast}_{\uparrow}[Obj\ Cell]\ \mathbf{pack}\ [\{x : Int\}, (\{x = 0\}, f)]\ \mathbf{as}\ CellT))\ \mathbf{as}\ [X, (s, m)]\ \mathbf{in} \\
&\qquad \mathbf{pack}\ [X, ((\mathsf{cast}_{\downarrow}\ (m\ s)).bump, m)]\ \mathbf{up}\ (Obj\ Cell)\ ) \Leftarrow get \\
&\quad \hookrightarrow (\ \mathbf{unpack}\ (\mathbf{pack}\ [\{x : Int\}, (\{x = 0\}, f)]\ \mathbf{as}\ CellT)\ \mathbf{as}\ [X, (s, m)]\ \mathbf{in} \\
&\qquad \mathbf{pack}\ [X, ((\mathsf{cast}_{\downarrow}\ (m\ s)).bump, m)]\ \mathbf{up}\ (Obj\ Cell)\ ) \Leftarrow get \\
&\quad \hookrightarrow (\mathbf{pack}\ [\{x : Int\}, ((\mathsf{cast}_{\downarrow}\ (f\ \{x = 0\})).bump, f)]\ \mathbf{up}\ (Obj\ Cell)) \Leftarrow get \\
&\quad = \mathbf{unpack}\ (\mathsf{cast}_{\downarrow}\ (\mathbf{pack}\ [\{x : Int\}, ((\mathsf{cast}_{\downarrow}\ (f\ \{x = 0\})).bump, f)]\ \mathbf{up}\ (Obj\ Cell))) \\
&\qquad \mathbf{as}\ [X, (s, m)]\ \mathbf{in}\ (\mathsf{cast}_{\downarrow}\ (m\ s)).get \\
&\quad = \mathbf{unpack}\ (\mathsf{cast}_{\downarrow}\ \mathsf{cast}_{\uparrow}[Obj\ Cell]\ (\mathbf{pack}\ [\{x : Int\}, ((\mathsf{cast}_{\downarrow}\ (f\ \{x = 0\})).bump, f)]\ \mathbf{as}\ CellT)) \\
&\qquad \mathbf{as}\ [X, (s, m)]\ \mathbf{in}\ (\mathsf{cast}_{\downarrow}\ (m\ s)).get \\
&\quad \hookrightarrow \mathbf{unpack}\ (\mathbf{pack}\ [\{x : Int\}, ((\mathsf{cast}_{\downarrow}\ (f\ \{x = 0\})).bump, f)]\ \mathbf{as}\ CellT) \\
&\qquad \mathbf{as}\ [X, (s, m)]\ \mathbf{in}\ (\mathsf{cast}_{\downarrow}\ (m\ s)).get \\
&\quad \hookrightarrow (\mathsf{cast}_{\downarrow}(f\ ((\mathsf{cast}_{\downarrow}\ (f\ \{x = 0\})).bump))).get \\
&\quad \hookrightarrow (\mathsf{cast}_{\downarrow}(f[(\mathsf{cast}_{\downarrow}\ (f\ \{x = 0\})).bump])).get \\
&\quad \hookrightarrow \{\ get = ((\mathsf{cast}_{\downarrow}\ (f\ \{x = 0\})).bump).x,\ set = \lambda n : Int.\ \{x = n\}, \\
&\qquad bump = \{x = ((\mathsf{cast}_{\downarrow}\ (f\ \{x = 0\})).bump).x + 1\}\ \}.get \\
&\quad \hookrightarrow ((\mathsf{cast}_{\downarrow}\ (f\ \{x = 0\})).bump).x \\
&\quad \hookrightarrow ((\mathsf{cast}_{\downarrow}\ f[\{x = 0\}]).bump).x \\
&\quad \hookrightarrow (\{\ get = \{x = 0\}.x,\ set = \lambda n : Int.\ \{x = n\}, \\
&\qquad bump = \{x = \{x = 0\}.x + 1\}\ \}.bump).x \\
&\quad \hookrightarrow \{x = \{x = 0\}.x + 1\}.x \hookrightarrow \cdots \hookrightarrow 1
\end{aligned}
$$

Figure 5.1. Evaluation of $(c \Leftarrow bump) \Leftarrow get$

simple pattern matching on pairs for brevity. The hidden state is unpacked as s with type X. The function containing methods is m with type $X \to I\ X$. The record of methods can be obtained by applying m to s. Noting that the subtyping relation $I\ X \leqslant Cell\ X$ holds, the type of $m\ s$ can be converted from $I\ X$ to $Cell\ X$ by subsumption. Another $\mathsf{cast}_{\downarrow}$ further reduces $Cell\ X$ to record type for accessing the member get. The encoding of message $bump$ is similar but needs to repack the resulting object:

$$
\begin{aligned}
c \Leftarrow bump \triangleq\ & bumpM\ Cell\ c \\
bumpM\ =\ & \lambda I \leqslant Cell : \star \to \star.\ \lambda o : Obj\ I. \\
& \mathbf{unpack}\ (\mathsf{cast}_{\downarrow}\ o)\ \mathbf{as}\ [X, (s, m)]\ \mathbf{in} \\
& \mathbf{pack}\ [X, ((\mathsf{cast}_{\downarrow}\ (m\ s)).bump, m)]\ \mathbf{up}\ (Obj\ I)
\end{aligned}
$$

since the $bump$ method returns a record but not an object. The extra **pack** here is required to create a new object using the result of $bump$ as the new hidden state.

Similarly to the original example [Bruce et al. 1999], we can examine the encoding by evaluating the expression $(c \Leftarrow bump) \Leftarrow get$ using call-by-name reduction ($\hookrightarrow$). In Figure 5.1, we show the evaluation steps of the expression $(c \Leftarrow bump) \Leftarrow get$ using call-by-name reduction ($\hookrightarrow$). The evaluation result is 1 as expected, where f is the function containing the definition of methods:

$$
\begin{aligned}
f = \lambda s : \{x : Int\}.\ & \mathsf{cast}_{\uparrow}\ [Cell\ \{x : Int\}] \\
& \{get = s.x,\ set = \lambda n : Int.\ \{x = n\},\ bump = \{x = s.x + 1\}\}
\end{aligned}
$$

and $f[r]$ is one-step reduction of the application $f\ r$:

$$f[r] = \mathsf{cast}_{\uparrow}\ [Cell\ \{x : Int\}] \\ \{get = r.x,\ set = \lambda n : Int.\ \{x = n\},\ bump = \{x = r.x + 1\}\}$$

Note that we assume reduction rules for records, existential packages and integer addition are available in the call-by-name reduction relation ($\hookrightarrow$). We skip steps for desugaring the message sending operation, e.g., $o \Leftarrow get = getM[Cell][o]$. We emphasize that the object encoding example here requires the additional feature of $\lambda I_{\leqslant}$ over PITS, i.e., higher-order subtyping, which is essential for encoding generic message functions.

5.2 The $\lambda I_{\leqslant}$ Calculus

We present the $\lambda I_{\leqslant}$ calculus in this section. The calculus features a unified syntax with only one syntactic level, and it is based on the λI calculus [Yang et al. 2016], a specific PITS with a single sort $\star$ and the "type-in-type" axiom (see Section 3.2.1). The novelty over λI/PITS is subtyping. To integrate subtyping, typing is unified with the subtyping relation. Thus the typing relation can be viewed as a special case of subtyping. We demonstrate the syntax, operational and static semantics of $\lambda I_{\leqslant}$ in the rest of this section. Notice that $\lambda I_{\leqslant}$ discussed in this section does not contain recursion, which can be supported by following PITS. We leave the discussion of recursion to Section 5.6.

5.2.1 Syntax

Figure 5.2 shows the syntax of expressions in $\lambda I_{\leqslant}$. It follows the unified syntax of Pure Type Systems [Barendregt 1992] where terms, types and a single kind $\star$ are defined in the same syntactic category. By convention, we still use different metavariables to indicate if expressions are terms (e) or types (A, B, C, etc.).

Cast Operators. As in PITS, cast operators $\mathsf{cast}_{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ are used for explicit type-level computation in $\lambda I_{\leqslant}$. In particular, $\mathsf{cast}_{\downarrow}$ and $\mathsf{cast}_{\uparrow}$ convert the type of an expression by a one-step reduction or expansion, respectively. $\mathsf{cast}_{\uparrow}$ needs to be annotated with the result type of one-step expansion, while $\mathsf{cast}_{\downarrow}$ does not, since one-step reduction is deterministic (see Section 5.3.4).

Bounded Quantification. Functions are written as $\lambda x \leqslant e_1 : A.\ e_2$, which support *bounded quantification* as in System $F_{\leqslant}$ [Cardelli et al. 1994]. The bound term e_1 is annotated with a type A. Correspondingly, function types written as $\Pi x \leqslant e : A.\ B$ also contain a bound term e. Function types can be *dependent* if x occurs free in B. The top type $\top$ is a supertype of any well-formed term, e.g., $3 \leqslant \top$. The top type generalizes the conventional top type in System $F_{\leqslant}$, which is only a supertype of well-formed types, e.g., $Int \leqslant \top$.

Syntactic Sugar. Unbounded functions ($\lambda x : A.\ e$) and function types ($\Pi x : A.\ B$) are not defined as primitives in the syntax. With the generalized top type, we can define them as syntactic sugar of *top-bounded* ones, i.e., $\lambda x \leqslant \top : A.\ e$ and $\Pi x \leqslant \top : A.\ B$ as shown in Figure 5.2. We also treat arrow types $A \to B$ as syntactic sugar of $\Pi x : A.\ B$ if x does not occur free in B.

Expressions	e, A, B	$::=$	$x \mid \star \mid \top \mid e_1\ e_2 \mid \mathsf{cast}_{\uparrow}\ [A]\ e \mid \mathsf{cast}_{\downarrow}\ e$
		$\mid$	$\lambda x \leqslant e_1 : A.\ e_2 \mid \Pi x \leqslant e : A.\ B$
Contexts	Γ	$::=$	$\varnothing \mid \Gamma, x \leqslant e : A$
Inert Terms	u	$::=$	$x \mid \top \mid u\ e \mid \mathsf{cast}_{\downarrow}\ u$
Values	v	$::=$	$\star \mid \lambda x \leqslant e_1 : A.\ e_2 \mid \Pi x \leqslant e : A.\ B \mid \mathsf{cast}_{\uparrow}\ [A]\ e \mid u$
Syntactic Sugar	$\lambda x : A.\ e$	$\triangleq$	$\lambda x \leqslant \top : A.\ e$
	$\Pi x : A.\ B$	$\triangleq$	$\Pi x \leqslant \top : A.\ B$
	$A \to B$	$\triangleq$	$\Pi x : A.\ B$ where $x \notin \mathsf{FV}(B)$

Figure 5.2. Syntax

Context. The syntax of context Γ is defined in Figure 5.2. The variable binding only has the bounded form $x \leqslant e : A$ where the bound term e has type A. Similar to the treatment of unbounded functions above, we can treat an unbounded variable binding as the syntactic sugar of a top-bounded binding, i.e., $\Gamma, x : A \triangleq \Gamma, x \leqslant \top : A$.

5.2.2 Operational Semantics

Figure 5.3 shows the definition of one-step reduction ($\hookrightarrow$), which is used for both evaluation and type conversion (via cast operators). It follows the *call-by-name* style and is *weak-head*. R-Beta performs the beta reduction and does not require the argument to be a value. R-CastElim cancels consecutive $\mathsf{cast}_{\downarrow}$ and $\mathsf{cast}_{\uparrow}$. R-App and R-CastDn perform reduction at the head term of an application and the inner term of $\mathsf{cast}_{\downarrow}$, respectively.

Since the reduction relation ($\hookrightarrow$) is also used for type conversion, we may encounter *open* terms during reduction. However, some open terms are *stuck* terms that are not reducible by $\hookrightarrow$. For example, an application starting with a variable: $x\ e_1\ e_2\ \ldots\ e_n$. Also, as the top type is generalized, assuming it is a supertype of an n-ary function, we can have a well-formed but stuck term such as $\top\ e_1\ e_2\ \ldots\ e_n$. Furthermore, if we replace x and $\top$ in both stuck terms by $\mathsf{cast}_{\downarrow}\ x$ and $\mathsf{cast}_{\downarrow}\ \top$ respectively, they still cannot be reduced.

We introduce a syntactic category called *inert terms* to cover such stuck terms. The terminology is inspired by the fireball calculus [Paolini and Della Rocca 1999; Accattoli and Guerrieri 2016]. Figure 5.2 shows the definition of inert terms, ranged over by metavariable u. Two base inert terms are variables and the top type. Compound inert terms are either an application leading with an inert term, i.e., $u\ e$, or down-cast inert term, i.e., $\mathsf{cast}_{\downarrow}\ u$. We treat inert terms as values. Figure 5.2 shows the syntax of values, ranged over by metavariable v, as shown in Figure 5.2. A value can either be the kind $\star$, a function, a function type, a $\mathsf{cast}_{\uparrow}$ term or an inert term.

There are several alternative designs on reduction rules and syntax of values, e.g., beta-top ($\beta\top$) reduction [Pierce and Steffen 1997] and $\mathsf{cast}_{\uparrow}\ [A]\ v$ as a value [Pierce 2002; Yang et al. 2016]. We will discuss these designs and their trade-offs later in Section 5.6.

5.2.3 Static Semantics

Figure 5.4 shows the rules of static semantics, including two judgment forms: context well-formedness $\vdash \Gamma$ and unified subtyping $\Gamma \vdash e_1 \leqslant e_2 : A$. The unified subtyping judgment $\Gamma \vdash e_1 \leqslant e_2 : A$ serves as both subtyping and typing judgment. It can be interpreted as "e_1 is a subtype of e_2 and both of them have type A". The inference rules are developed to satisfy such interpretation. For brevity, if e_1 and e_2 are the same (i.e. $e_1 = e_2 = e$), we use the syntactic

$\boxed{e_1 \hookrightarrow e_2}$ *(Call-by-name Reduction)*

$$\textsc{R-Beta}\ \frac{}{(\lambda x : A.\ e_1)\ e_2 \hookrightarrow e_1[x \mapsto e_2]} \qquad \textsc{R-App}\ \frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \textsc{R-Mu}\ \frac{}{\mu x : A.\ e \hookrightarrow e[x \mapsto \mu x : A.\ e]}$$

$$\textsc{R-CastDn}\ \frac{e \hookrightarrow e'}{\mathsf{cast}_{\downarrow}\ e \hookrightarrow \mathsf{cast}_{\downarrow}\ e'} \qquad \textsc{R-CastElim}\ \frac{}{\mathsf{cast}_{\downarrow}\ (\mathsf{cast}_{\uparrow}\ [A]\ e) \hookrightarrow e}$$

Figure 5.3. Operational semantics

sugar $\Gamma \vdash e : A$ (see Figure 5.4), which also has the same form of typing judgment in traditional systems. We also use $\Gamma \vdash A : \star$ to check if type A is well-formed, i.e., has the kind $\star$. Thus in $\lambda I_{\leqslant}$, subtyping, typing and well-formedness of types are all unified by the unified subtyping judgment:

Unified Subtyping		$\Gamma \vdash e_1 \leqslant e_2 : A$
Typing	$\Gamma \vdash e : A$	$\triangleq\ \Gamma \vdash e \leqslant e : A$
Well-formed Types	$\Gamma \vdash A : \star$	$\triangleq\ \Gamma \vdash A \leqslant A : \star$

A key benefit of unified subtyping is that the mutual dependency issue between typing and subtyping found in many traditional higher-order subtyping systems can be avoided since typing is just a special case of subtyping.

The context well-formedness judgment $\vdash \Gamma$ is defined inductively on the structure of Γ. Whenever adding a fresh binding $x \leqslant e : A$ to the context Γ, the judgment ensures e has a well-formed type A.

We briefly introduce the basic rules and discuss the rest in detail. S-Ax defines the reflexivity of the kind $\star$ and follows the "type-in-type" axiom [Cardelli 1986b] for the typing of $\star$. S-VarRefl defines the reflexivity of a variable and its typing by looking up the context. S-VarTrans defines the variable lookup followed by transitivity, which follows the algorithmic version of System $F_{\leqslant}$ [Curien and Ghelli 1992].

Generalized Top Type. S-Top defines subtyping for the generalized top type: a supertype of any term e which has the same type A as e. A special case is when e is also a top type. For this case we need to define the reflexivity of top type as in the rule S-TopRefl, which indicates that the top type can have any well-formed type A. In other words, any well-formed type can be inhabited by the generalized top type, which causes *logical inconsistency*. Note that allowing "type-in-type" axiom in S-Ax already brings logical inconsistency [Barendregt 1992]. Our goal is to investigate the calculus for traditional programming that allows general recursion, which is logically inconsistent any way. Thus, we do not consider generalized top type or "type-in-type" axiom problematic. With top type generalized, bounded and unbounded quantification are unified, which significantly simplifies the system.

Functions and Function Types. S-Abs defines the subtyping relation between functions, which follows the *invariant* rule for type operators in System $F^{\omega}_{\leqslant}$ [Pierce and Steffen 1997]. It requires the bounds and argument types being compared to be identical. The first line of premises in S-Abs checks the well-formedness of binding. The second line of premises checks if the function bodies are covariant and their type is well-formed.

$\boxed{\vdash \Gamma}$ Context Well-formedness

$$\textsc{WU-Empty}\ \frac{}{\vdash \varnothing} \qquad \textsc{WU-Cons}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A : \star}{\vdash \Gamma, x \leqslant e : A}$$

$\boxed{\Gamma \vdash e_1 \leqslant e_2 : A}$ *(Unified Subtyping)*

$$\textsc{S-Ax}\ \frac{\vdash \Gamma}{\Gamma \vdash \star \leqslant \star : \star} \qquad \textsc{S-VarRefl}\ \frac{\vdash \Gamma \qquad x \leqslant e : A \in \Gamma}{\Gamma \vdash x \leqslant x : A} \qquad \textsc{S-VarTrans}\ \frac{x \leqslant e_1 : A \in \Gamma \qquad \Gamma \vdash e_1 \leqslant e_2 : A}{\Gamma \vdash x \leqslant e_2 : A}$$

$$\textsc{S-Top}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash e \leqslant \top : A} \qquad \textsc{S-TopRefl}\ \frac{\Gamma \vdash A : \star}{\Gamma \vdash \top \leqslant \top : A}$$

$$\textsc{S-Abs}\ \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash A : \star \qquad \Gamma, x \leqslant e_1 : A \vdash e_2 \leqslant e_2' : B \qquad \Gamma, x \leqslant e_1 : A \vdash B : \star}{\Gamma \vdash (\lambda x \leqslant e_1 : A.\ e_2) \leqslant (\lambda x \leqslant e_1 : A.\ e_2') : \Pi x \leqslant e_1 : A.\ B}$$

$$\textsc{S-App}\ \frac{\Gamma \vdash e_1 \leqslant e_2 : \Pi x \leqslant e_3 : B.\ C \qquad \Gamma \vdash A \leqslant e_3 : B}{\Gamma \vdash e_1\ A \leqslant e_2\ A : C[x \mapsto A]}$$

$$\textsc{S-Prod}\ \frac{\Gamma \vdash A' \leqslant A : \star \qquad \Gamma \vdash e : A' \qquad \Gamma \vdash A : \star \qquad \Gamma, x \leqslant e : A \vdash B : \star \qquad \Gamma, x \leqslant e : A' \vdash B \leqslant B' : \star}{\Gamma \vdash (\Pi x \leqslant e : A.\ B) \leqslant (\Pi x \leqslant e : A'.\ B') : \star}$$

$$\textsc{S-CastUp}\ \frac{\Gamma \vdash B : \star \qquad \Gamma \vdash e_1 \leqslant e_2 : A \qquad B \hookrightarrow A}{\Gamma \vdash \mathsf{cast}_{\uparrow}\ [B]\ e_1 \leqslant \mathsf{cast}_{\uparrow}\ [B]\ e_2 : B}$$

$$\textsc{S-CastDn}\ \frac{\Gamma \vdash B : \star \qquad \Gamma \vdash e_1 \leqslant e_2 : A \qquad A \hookrightarrow B}{\Gamma \vdash \mathsf{cast}_{\downarrow}\ e_1 \leqslant \mathsf{cast}_{\downarrow}\ e_2 : B} \qquad \textsc{S-Sub}\ \frac{\Gamma \vdash e_1 \leqslant e_2 : A \qquad \Gamma \vdash A \leqslant B : \star}{\Gamma \vdash e_1 \leqslant e_2 : B}$$

Syntactic Sugar $\quad \Gamma \vdash e : A \triangleq \Gamma \vdash e \leqslant e : A$

Figure 5.4. Static semantics

S-Prod defines the relation between function types. Unlike S-Abs, it only requires the bounds to be identical. The argument types can vary and are *contravariant*. Such design follows the *Kernel Fun* variant [Cardelli and Wegner 1985] of System $F_{\leqslant}$. S-Prod can be viewed as a combination of the subtyping rules for arrow types and universal types of System $F_{\leqslant}$:

$$\textsc{FS-Arrow}\ \frac{\Delta \vdash T_1 \leqslant U_1 \qquad \Delta \vdash U_2 \leqslant T_2}{\Delta \vdash (U_1 \to U_2) \leqslant (T_1 \to T_2)} \qquad \textsc{FS-Forall}\ \frac{\Delta, X \leqslant U \vdash T_1 \leqslant T_2}{\Delta \vdash (\forall X \leqslant U.\ T_1) \leqslant (\forall X \leqslant U.\ T_2)}$$

The first premise of S-Prod checks the contravariance of argument types, similar to the rule for arrow types. The last premise checks the covariance of co-domains of function types with bound fixed, similar to the rule for universal types. Other premises check the well-formedness.

Pointwise Subtyping. S-App defines subtyping between applications and uses a *pointwise* subtyping rule originated from System $F^{\omega}_{\leqslant}$ [Pierce and Steffen 1997], which is also used in many systems with higher order subtyping [Hutchins 2010; Aspinall and Compagnoni 1996; Zwanenburg 1999]. When comparing two applications, we require the arguments to be identical and only compare the head terms, equivalently to type operators in $F^{\omega}_{\leqslant}$. The first premise of S-App ensures the head term to have a function type, e.g., $\Pi x \leqslant e_3 : B.\ C$. The second premise checks the bound and typing requirements: if the argument A is a subtype of e_3 and A has the type B.

Explicit Casts and Syntactic Equality. S-CastUp and S-CastDn are rules for explicit cast operators. They can be seen as a generalization of typing rules of fold and unfold from iso-recursive types (see Section 4.1.4). Weak-head reduction ($\hookrightarrow$) is used for type-level conversion. Note that when comparing $\mathsf{cast}_{\uparrow}$ terms, we require the annotations to be the same. S-Sub is the subsumption rule. The second premise checks the subtyping relation between well-formed types by reusing the unified subtyping judgment. Note that S-Sub does *not* subsume the implicit conversion rule, which can be found in $F^{\omega}_{\leqslant}$ and Pure Type Systems. Because the unified subtyping judgment does not subsume beta conversion, i.e., $(\lambda x : \star.\ x)\ Int \leqslant Int$ does not hold. As a consequence, types of expressions are equal only up to syntactic equality (i.e. alpha equality), but not beta equality. Nevertheless, we can recover type-level computation through explicit cast operators.

Algorithmic up to Subtyping. The unified subtyping rules shown in Figure 5.4 are *declarative* because of the subsumption rule S-Sub. But the system is *almost* algorithmic: if we ignore the typing result and only consider the subtyping part, the system becomes algorithmic. Like the algorithmic version of System $F_{\leqslant}$, there is no built-in transitivity rule defined in $\lambda I_{\leqslant}$. Actually, transitivity can be proved from other rules (see Section 5.3.2).

5.3 The Metatheory of Unified Subtyping

In this section, we discuss the metatheory of $\lambda I_{\leqslant}$ by focusing on two main targets: transitivity and type safety. We emphasize here that in previous work the metatheory for the combination between dependent types and subtyping was a key difficulty, greatly due to the entanglement between the metatheory of subtyping and typing. With unified subtyping we develop a single metatheory for the new relation instead. Traditional theorems related to the metatheory of typing and subtyping can then be viewed as particular instantiations of the unified subtyping theorems. Because the unified subtyping relation is new, working out the metatheory for our system actually required figuring out which theorems to prove (i.e. what form should they have); and in which order to prove them. It is crucial (and non-trivial) to prove the *right* theorems in the correct *order*. Nevertheless, once the form of the theorems and the order in which they should be proved are set, then the proofs can actually be done with simple techniques similar to those used in more traditional systems. The dependency diagram of main lemmas in this section is shown in Figure 5.5.

5.3.1 Basic Lemmas

Before going to the proof of transitivity, we first discuss several important basic lemmas including reflexivity, weakening, consistency of typing and narrowing.

Reflexivity. The subtyping relation in System $F_{\leqslant}$ is reflexive, i.e., $\Delta \vdash T \leqslant T$ holds for any *well-formed* type T and context Δ. Since unified subtyping in $\lambda I_{\leqslant}$ tracks typing results, the relation in reflexive form, i.e., $\Gamma \vdash e \leqslant e : A$, works like a typing judgment $\Gamma \vdash e : A$ (recall the syntactic sugar in Figure 5.4). Reflexivity does not hold for arbitrary e and A. However reflexivity does hold for any well-(sub)typed terms. That is:

Lemma 5.3.1 (Reflexivity). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$, *then both* $\Gamma \vdash e_1 : A$ *and* $\Gamma \vdash e_2 : A$ *hold.*

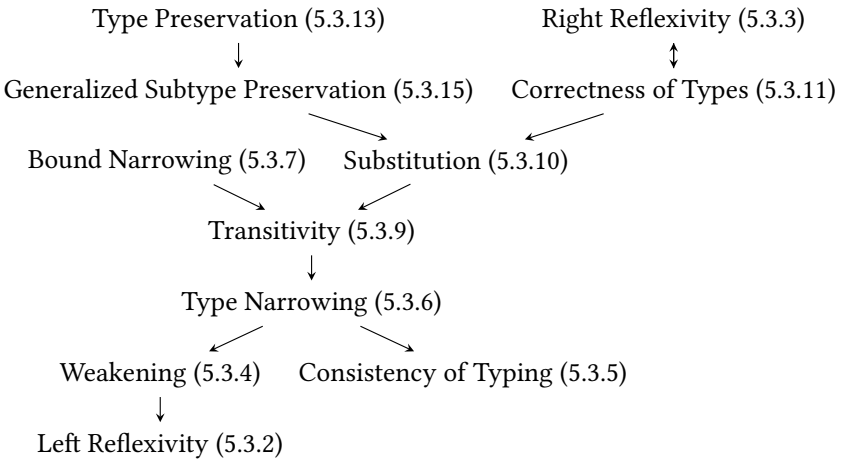

Figure 5.5. Dependency of lemmas for the metatheory of unified subtyping

This lemma is also called *validity* in some literature [Abel and Rodriguez 2008]. Here we call it "reflexivity" because conclusions are still (unified) subtyping relations in reflexive form. It also meets the interpretation of unified subtyping mentioned in Section 5.2.3. We separate the reflexivity lemma into two sub-lemmas by dividing the conclusion:

Lemma 5.3.2 (Left Reflexivity). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$*, then* $\Gamma \vdash e_1 : A$ *holds.*

Lemma 5.3.3 (Right Reflexivity). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$*, then* $\Gamma \vdash e_2 : A$ *holds.*

Left reflexivity can be proved by induction on the derivation of $\Gamma \vdash e_1 \leqslant e_2 : A$. However, right reflexivity is difficult to prove due to the generalized top type. Consider the case of S-Top, i.e., $\Gamma \vdash e \leqslant \top : A$. We know $\Gamma \vdash e : A$ from the premise. The target $\Gamma \vdash \top : A$ requires A to be well-formed, i.e., $\Gamma \vdash A : \star$, as indicated by the premise of S-TopRefl. To prove $\Gamma \vdash A : \star$ from $\Gamma \vdash e : A$, we need a lemma called *correctness of types* (Lemma 5.3.11), which is not available currently. We will show the full proof later in Section 5.3.3. Currently without right reflexivity, we add redundant premises in typing rules to simplify the proofs. For example, in rule S-Prod, the third premise $\Gamma \vdash A : \star$ is derivable from the first premise $\Gamma \vdash A' \leqslant A : \star$ by right reflexivity. Once right reflexivity is shown, such additional premises can be removed without changing the type system.

Weakening. The weakening lemma is standard:

Lemma 5.3.4 (Weakening). *If* $\Gamma_1, \Gamma_3 \vdash e_1 \leqslant e_2 : A$ *and* $\vdash \Gamma_1, \Gamma_2, \Gamma_3$*, then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash e_1 \leqslant e_2 : A$.

The proof is by induction on the derivation of $\Gamma_1, \Gamma_3 \vdash e_1 \leqslant e_2 : A$. The only interesting case is when S-Prod is the last derivation. The last premise of S-Prod adds binding $x \leqslant e : A'$ into the context Γ. We need to ensure A' is well-formed, i.e., $\Gamma \vdash A' : \star$, as required by context well-formedness. Though not included in the premise, it can be derived by applying left reflexivity (Lemma 5.3.2) to the first premise, i.e., $\Gamma \vdash A' \leqslant A : \star$.

Consistency of Typing. We prove a simple yet important lemma, called *consistency of typing*:

Lemma 5.3.5 (Consistency of Typing). *If* $\Gamma \vdash e_1 : A$ *and* $\Gamma \vdash e_1 \leqslant e_2 : B$, *then* $\Gamma \vdash e_1 \leqslant e_2 : A$.

The proof is by induction on the derivation of $\Gamma \vdash e_1 \leqslant e_2 : B$. This lemma is the key to decoupling typing from unified subtyping. To prove $\Gamma \vdash e_1 \leqslant e_2 : A$, we can individually show 1) e_1 has the type A and 2) e_1 is a subtype of e_2 regardless of typing, as long as there is some type B such that $\Gamma \vdash e_1 \leqslant e_2 : B$.

Narrowing. We have two narrowing lemmas in $\lambda I_{\leqslant}$, type narrowing and bound narrowing:

Lemma 5.3.6 (Type Narrowing). *Given* $\Gamma_1, x \leqslant e : B, \Gamma_2 \vdash e_1 \leqslant e_2 : C$, *if* $\Gamma_1 \vdash A \leqslant B : \star$ *and* $\Gamma_1 \vdash e : A$, *then* $\Gamma_1, x \leqslant e : A, \Gamma_2 \vdash e_1 \leqslant e_2 : C$.

Lemma 5.3.7 (Bound Narrowing). *If* $\Gamma_1, x \leqslant e : B, \Gamma_2 \vdash e_1 \leqslant e_2 : C$ *and* $\Gamma_1 \vdash e' \leqslant e : B$, *then* $\Gamma_1, x \leqslant e' : B, \Gamma_2 \vdash e_1 \leqslant e_2 : C$.

As indicated by the name, for a binding $x \leqslant e : B$ in the context, type narrowing changes its type from B to a subtype A, while bound narrowing changes its bound from e to a subtype e'. We only prove type narrowing here, since bound narrowing depends on transitivity, as will be discussed later in Section 5.3.3. The type narrowing lemma is proved by induction on the derivation of $\Gamma_1, x \leqslant e : B, \Gamma_2 \vdash e_1 \leqslant e_2 : C$. The only interesting case is when the last derivation uses S-VarTrans, i.e., e_1 is a variable. It is easy to prove by the induction hypothesis when e_1 is not x. When $e_1 = x$, we know $B = C$ and our target is to show $\Gamma_1, x \leqslant e : A, \Gamma_2 \vdash x \leqslant e_2 : B$. By applying the subsumption rule S-Sub and S-VarTrans, our target becomes $\Gamma_1, x \leqslant e : A, \Gamma_2 \vdash e \leqslant e_2 : A$. Note that we have $\Gamma_1, x \leqslant e : A, \Gamma_2 \vdash e \leqslant e_2 : B$ by the induction hypothesis. The only gap is the typing result, which should be A but not B. Thus, we can apply the consistency of typing lemma (Lemma 5.3.5) and prove $\Gamma_1, x \leqslant e : A, \Gamma_2 \vdash e : A$ instead, which is immediate by weakening (Lemma 5.3.4).

5.3.2 Transitivity

Transitivity is a desirable property of systems with subtyping. Declarative presentations of calculi often include a built-in transitivity rule:

$$\frac{\Gamma \vdash e_1 \leqslant e_2 : A \qquad \Gamma \vdash e_2 \leqslant e_3 : A}{\Gamma \vdash e_1 \leqslant e_3 : A}\ \text{S-Trans}$$

This simplifies the proof of lemmas such as narrowing and substitution. However, noticing that e_1 and e_3 can be in any form, the rule can be applied any time during derivation, which complicates the inversion of subtyping judgments. A process called *transitivity elimination* [Pierce 2002; Pierce and Steffen 1997; Compagnoni 1995] can be used to avoid such complexity brought by the transitivity rule. The declarative system is reformulated into an algorithmic one without a transitivity rule. The transitivity property is then proved against the algorithmic system. Similarly, we formulate $\lambda I_{\leqslant}$ without a built-in transitivity rule but only with a base case for variables (i.e. S-VarTrans), as mentioned in Section 5.2.3. Next we show the proof of transitivity in $\lambda I_{\leqslant}$.

First, we need to generalize the form of transitivity. The form of rule S-Trans is too restricted: conditions are required to have the same type. This causes issues when both conditions are

derived from S-Sub:

$$\dfrac{\dfrac{\text{S-Sub}}{\dfrac{\Gamma \vdash e_1 \leqslant e_2 : B_1 \qquad \Gamma \vdash B_1 \leqslant A : \star}{\Gamma \vdash e_1 \leqslant e_2 : A}} \qquad \dfrac{\text{S-Sub}}{\dfrac{\Gamma \vdash e_2 \leqslant e_3 : B_2 \qquad \Gamma \vdash B_2 \leqslant A : \star}{\Gamma \vdash e_2 \leqslant e_3 : A}}}{\Gamma \vdash e_1 \leqslant e_3 : A}\ \text{S-Trans}$$

We only know B_1 and B_2 are both subtypes of A but cannot determine the relation between them. The induction hypothesis cannot be applied since it requires B_1 and B_2 to be the same. Thus, we generalize the property into

$$\dfrac{\Gamma \vdash e_1 \leqslant e_2 : A \qquad \Gamma \vdash e_2 \leqslant e_3 : B}{\Gamma \vdash e_1 \leqslant e_3 : A}\ \text{S-Trans2}$$

where the conditions are allowed to have different types and the conclusion needs to have the same type as the first condition. The proof of the generalized transitivity is standard [Pierce 2002] by induction on the size of e_2 and an inner induction on the derivation of the first condition $\Gamma \vdash e_1 \leqslant e_2 : A$. We only discuss the interesting case when both derivations end with S-Prod. We have $e_1 = \Pi x \leqslant e : A_1.\ B_1$, $e_2 = \Pi x \leqslant e : A_2.\ B_2$, and $e_3 = \Pi x \leqslant e : A_3.\ B_3$, with

$$\Gamma \vdash A_2 \leqslant A_1 : \star \quad (1) \qquad \Gamma, x \leqslant e : A_2 \vdash B_1 \leqslant B_2 : \star \quad (2)$$
$$\Gamma \vdash A_3 \leqslant A_2 : \star \quad (3) \qquad \Gamma, x \leqslant e : A_3 \vdash B_2 \leqslant B_3 : \star \quad (4)$$

For clarity, we omit all derivations for well-formedness checking in the discussion, which can be trivially proved by the induction hypothesis. Our target is to prove $\Gamma \vdash A_3 \leqslant A_1 : \star$ and $\Gamma, x \leqslant e : A_3 \vdash B_1 \leqslant B_3 : \star$. The first target can be obtained by combining (1) and (3) using the outer induction hypothesis since A_2 has smaller size than e_2. Noting that the context of the (2) is different from (4) and the second target, we use Lemma 5.3.6 to narrow the type of the binding to obtain $\Gamma, x \leqslant e : A_3 \vdash B_1 \leqslant B_2 : \star$. Then we can similarly obtain the second target by the outer induction hypothesis since the size of B_2 is smaller than e_2. We conclude the generalized transitivity by the following lemma:

Lemma 5.3.8 (Generalized Transitivity)**.** *If* $\Gamma \vdash e_1 \leqslant e_2 : A$ *and* $\Gamma \vdash e_2 \leqslant e_3 : B$*, then* $\Gamma \vdash e_1 \leqslant e_3 : A$.

Thus, the original transitivity is an immediate corollary:

Lemma 5.3.9 (Transitivity)**.** *If* $\Gamma \vdash e_1 \leqslant e_2 : A$ *and* $\Gamma \vdash e_2 \leqslant e_3 : A$*, then* $\Gamma \vdash e_1 \leqslant e_3 : A$.

As shown in Figure 5.5, the proof of generalized transitivity depends on type narrowing (Lemma 5.3.6) and type narrowing depends on consistency of typing (Lemma 5.3.5). Actually, we can view consistency of typing as a special case of generalized transitivity by letting $e_1 = e_2 = e_1'$ and $e_3 = e_2'$. This indicates that type narrowing can also be proved using generalized transitivity. Thus, an alternative approach is to prove generalized transitivity and type narrowing simultaneously. A potential issue is that this approach makes these two lemmas mutually dependent. We choose to first prove a weaker version of generalized transitivity, i.e., consistency of typing, which has a much simpler proof. Then we can show type narrowing before transitivity without causing circularity.

5.3.3 Basic Lemmas, Revisited

Recall that in Section 5.3.1 we leave two lemmas unproved, i.e., right reflexivity (Lemma 5.3.3) and bound narrowing (Lemma 5.3.7), which depend on other lemmas that were not available yet. As we have proved transitivity in Section 5.3.2, we can recover the proof of these two lemmas.

Bound Narrowing. Similar to type narrowing (Lemma 5.3.6), bound narrowing (Lemma 5.3.7) is proved by induction on the derivation of $\Gamma_1, x \leqslant e : B, \Gamma_2 \vdash e_1 \leqslant e_2 : C$. We consider the interesting case when the derivation ends with S-VarTrans. If e_1 is not x, it is trivial to prove by the induction hypothesis. If $e_1 = x$, we have $B = C$ and our target is to show $\Gamma_1, x \leqslant e' : B, \Gamma_2 \vdash x \leqslant e_2 : B$. By the induction hypothesis, we have $\Gamma_1, x \leqslant e' : B, \Gamma_2 \vdash e \leqslant e_2 : B$. Noticing that by weakening (Lemma 5.3.4), we can obtain $\Gamma_1, x \leqslant e' : B, \Gamma_2 \vdash e' \leqslant e : B$ from the second condition. By transitivity (Lemma 5.3.9), we have $\Gamma_1, x \leqslant e' : B, \Gamma_2 \vdash e' \leqslant e_2 : B$. Also noticing that $x \leqslant e' : B \in \Gamma_1, x \leqslant e' : B, \Gamma_2$, we obtain the target by the rule S-VarTrans.

Substitution. We show that the substitution lemma holds in $\lambda I_{\leqslant}$:

Lemma 5.3.10 (Substitution). *If* $\Gamma_1, x \leqslant e : B, \Gamma_2 \vdash e_1 \leqslant e_2 : A$ *and* $\Gamma_1 \vdash e' \leqslant e : B$*, then* $\Gamma_1, \Gamma_2[x \mapsto e'] \vdash e_1[x \mapsto e'] \leqslant e_2[x \mapsto e'] : A[x \mapsto e']$.

The proof is standard by induction on the derivation of the first condition. It is similar to the proof of bound narrowing. Transitivity and weakening are also required for the case when S-TransVar is the last derivation. Note that the second condition $\Gamma_1 \vdash e' \leqslant e : B$ contains both subtyping requirement (e' is a subtype of e) and typing requirement (e' has type B). Thus, the substitution lemma in $\lambda I_{\leqslant}$ has only one form.

Right Reflexivity. As mentioned in Section 5.3.1, right reflexivity (Lemma 5.3.3) depends on correctness of types:

Lemma 5.3.11 (Correctness of Types). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$*, then* $\Gamma \vdash A : \star$.

But correctness of types also depends on right reflexivity. Consider the last derivation of $\Gamma \vdash e_1 \leqslant e_2 : A$ is S-Sub, where the premises are $\Gamma \vdash e_1 \leqslant e_2 : B$ and $\Gamma \vdash B \leqslant A : \star$. The conclusion $\Gamma \vdash A : \star$ holds if we apply right reflexivity to the second premise. Thus, we prove these two lemmas simultaneously by induction on the derivation of $\Gamma \vdash e_1 \leqslant e_2 : A$. Note that the proof of correctness of types also depends on the substitution lemma (Lemma 5.3.10) when the derivation ends with S-App.

With both left and right reflexivity proved, we conclude the reflexivity (Lemma 5.3.1) holds and the interpretation of unified subtyping in Section 5.2.3 is correct. One key insight here is that we do not prove the full reflexivity lemma first. Otherwise, it will cause circular dependency in the metatheory (imagine merging two nodes of left and right reflexivity in Figure 5.5).

5.3.4 Type Safety

We prove type safety by showing type preservation and progress lemmas [Wright and Felleisen 1994]. Though both lemmas have the same form as traditional systems, the typing judgment is just syntactic sugar of unified subtyping, as mentioned in Section 5.2.3.

Determinacy of Reduction. We first show that the one-step reduction relation is deterministic:

Lemma 5.3.12 (Determinacy of Reduction). *If* $e \hookrightarrow e_1$ *and* $e \hookrightarrow e_2$, *then* $e_1 = e_2$.

The proof is straightforward by induction on the derivation of $e \hookrightarrow e_1$. Note that the equality used in the conclusion is syntactic equality. The result of type-level reduction in the rule S-CastDn (i.e. type B) is unique. Thus, the $\mathsf{cast}_{\downarrow}$ term is not required to be annotated with the result type.

Type Preservation. Type preservation, also known as subject reduction [Wright and Felleisen 1994], states that reducing a term does not change its type:

Lemma 5.3.13 (Type Preservation). *If* $\Gamma \vdash e : A$ *and* $e \hookrightarrow e'$, *then* $\Gamma \vdash e' : A$.

However, if we try to directly prove this lemma by induction on the derivation of $\Gamma \vdash e \leqslant e : A$ (i.e. $\Gamma \vdash e : A$), the proof will get stuck. Consider the last derivation is S-CastDn and $e \hookrightarrow e'$ is an instance of R-CastElim with $e = \mathsf{cast}_{\downarrow}\,(\mathsf{cast}_{\uparrow}\,[B']\;e)$ and $e' = e$. We have $\Gamma \vdash \mathsf{cast}_{\uparrow}\,[B']\;e : B$ and $B \hookrightarrow A$. By inversion of S-CastUp, we can obtain $\Gamma \vdash e : A'$, $B' \hookrightarrow A'$ and $\Gamma \vdash B' \leqslant B : \star$. Our target is to show $\Gamma \vdash e : A$. If we can prove $\Gamma \vdash A' \leqslant A : \star$, then the target can be obtained immediately by the subsumption rule S-Sub. The relation is shown as follows:

$$\begin{array}{ccc} B' & \leqslant & B \\ \big\downarrow & & \big\downarrow \\ A' & \leqslant & A \end{array}$$

The subtyping relation in the second line requires a proof, which can be shown by the following lemma with a more general typing result other than the kind $\star$:

Lemma 5.3.14 (Subtype Preservation). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$, $e_1 \hookrightarrow e_1'$, $e_2 \hookrightarrow e_2'$, *then* $\Gamma \vdash e_1' \leqslant e_2' : A$.

We call this lemma *subtype preservation* indicating that the unified subtyping relation is preserved by reduction. Type preservation is just a special case of it when $e_1 = e_2 = e$ and $e_1' = e_2' = e'$. A naïve proof is by induction on the derivation of $\Gamma \vdash e_1 \leqslant e_2 : A$. The substitution lemma (Lemma 5.3.10) is required for the case when the derivation ends with S-App and both reductions are instances of R-Beta. However, the proof gets stuck when the derivation ends with S-CastDn, and both reductions are instances of R-CastElim with $e_1 = \mathsf{cast}_{\downarrow}\,(\mathsf{cast}_{\uparrow}\,[B]\;e_1')$ and $e_2 = \mathsf{cast}_{\downarrow}\,(\mathsf{cast}_{\uparrow}\,[B]\;e_2')$. The induction hypothesis does not work as it requires $\mathsf{cast}_{\uparrow}\,[B]\;e_1'$ and $\mathsf{cast}_{\uparrow}\,[B]\;e_2'$ to be reducible, while both of them are values (see Figure 5.2). To solve this issue, we need to generalize the subtype preservation lemma into the following one:

Lemma 5.3.15 (Generalized Subtype Preservation). *Given that* $\Gamma \vdash e_1 \leqslant e_2 : A$ *holds,*

1. *if both* e_1 *and* e_2 *are* $\mathsf{cast}_{\uparrow}$ *terms, i.e.,* $e_1 = \mathsf{cast}_{\uparrow}\,[B]\;e_1'$ *and* $e_2 = \mathsf{cast}_{\uparrow}\,[B]\;e_2'$, *and* $A \hookrightarrow A'$, $B \hookrightarrow B'$, *then* $\Gamma \vdash e_1' \leqslant e_2' : A'$;
2. *otherwise, if* $e_1 \hookrightarrow e_1'$ *and* $e_2 \hookrightarrow e_2'$, *then* $\Gamma \vdash e_1' \leqslant e_2' : A$.

Now the proof by induction can proceed with the generalized lemma. For the case which was stuck in the previous attempt, the conclusion is exactly the induction hypothesis that follows the case (1) of the lemma. The non-trivial case is when the derivation ends with the subsumption rule

S-Sub. When e_1 and e_2 are not both $\mathsf{cast}_\uparrow$ terms, the proof is trivial by the induction hypothesis. Otherwise, we have $e_1 = \mathsf{cast}_\uparrow\ [C]\ e_1'$ and $e_2 = \mathsf{cast}_\uparrow\ [C]\ e_2'$ such that

$$\Gamma \vdash \mathsf{cast}_\uparrow\ [C]\ e_1' \leqslant \mathsf{cast}_\uparrow\ [C]\ e_2' : B \quad (1) \qquad \Gamma \vdash B \leqslant A : \star \quad (2)$$
$$A \hookrightarrow A' \quad (3) \qquad C \hookrightarrow C' \quad (4)$$

Our target is to show $\Gamma \vdash e_1' \leqslant e_2' : A'$. Note that the annotations of $\mathsf{cast}_\uparrow$ in both terms must be the same (i.e. C) by S-CastUp. By inversion of (1), we have $\Gamma \vdash C \leqslant B : \star$. We first show there exists some B' such that $B \hookrightarrow B'$ by proving the following lemma:

Lemma 5.3.16 (Reduction Exists in the Middle)**.** *Given that* $\Gamma \vdash C \leqslant B : D$ *and* $\Gamma \vdash B \leqslant A : D$*, if* $C \hookrightarrow C'$ *and* $A \hookrightarrow A'$*, then there exists* B' *such that* $B \hookrightarrow B'$*.*

Then by induction hypothesis, we have $\Gamma \vdash e_1' \leqslant e_2' : B'$ from (1) by the first case of lemma and $\Gamma \vdash B' \leqslant A' : \star$ from (2) by the second (1st case impossible). Thus, we can prove the target by S-Sub and conclude Lemma 5.3.15. Finally, it is trivial to show that the original subtype preservation lemma is a corollary of the generalized one. Thus, we can conclude the type preservation lemma which is an immediate corollary of subtype preservation.

Progress. Progress states that well-formed terms do not get stuck:

Lemma 5.3.17 (Progress)**.** *If* $\varnothing \vdash e : A$ *then either* e *is a value* v *or there exists* e' *such that* $e \hookrightarrow e'$*.*

As we mentioned in Section 5.2.2, the type-level reduction in cast operators may encounter open terms. We prove a stronger progress lemma with a non-empty context:

Lemma 5.3.18 (Generalized Progress)**.** *If* $\Gamma \vdash e : A$ *then either* e *is a value* v *or there exists* e' *such that* $e \hookrightarrow e'$*.*

Then the original progress lemma is an immediate corollary of the stronger version. The proof is straightforward by induction on the derivation of $\Gamma \vdash e : A$. The definition of values is critical to the proof as it covers many stuck terms with variables and the top type (see also the discussion of inert terms in Section 5.2.2).

5.4 Algorithmic Version

As we mentioned in Section 5.2.3, the unified subtyping judgment presented in Figure 5.4 is declarative but almost algorithmic. The typing part is declarative because of the subsumption rule, while the subtyping part is algorithmic. If we separately check the typing part and subtyping part, we just need to develop an algorithm for type checking. We use *bidirectional type checking* [Pierce and Turner 2000], a standard technique to develop the type checking algorithm for type systems with subtyping. We show the soundness and completeness of the type and subtype checking algorithm with respect to the original unified subtyping judgment. Developing a unified algorithmic system is left as future work, as will be discussed in Section 8.2.

$\boxed{|e|}$ Erasure of Annotations

$$
\begin{aligned}
|x| &= x \\
|\star| &= \star \\
|\top| &= \top \\
|e_1\ e_2| &= |e_1|\ |e_2| \\
|\lambda x \leqslant e_1 : A.\ e_2| &= \lambda x \leqslant |e_1| : |A|\ .\ |e_2| \\
|\Pi x \leqslant e : A.\ B| &= \Pi x \leqslant |e| : |A|\ .\ |B| \\
|\mathsf{cast}_{\uparrow}\ [A]\ e| &= \mathsf{cast}_{\uparrow}\ [|A|]\ |e| \\
|\mathsf{cast}_{\downarrow}\ e| &= \mathsf{cast}_{\downarrow}\ |e| \\
|(e : A)| &= |e|
\end{aligned}
$$

$\boxed{|\Gamma|}$ Erasure of Annotations in Contexts

$$
\begin{aligned}
|\varnothing| &= \varnothing \\
|\Gamma, x \leqslant e : A| &= |\Gamma|\,, x \leqslant |e| : |A|
\end{aligned}
$$

Figure 5.6. Erasure of annotations

$\boxed{\Gamma \vdash e_1 \leqslant e_2}$ *(Algorithmic Subtyping)*

$$\textsc{AS-Star}\ \frac{}{\Gamma \vdash \star \leqslant \star} \qquad \textsc{AS-VarRefl}\ \frac{x \leqslant e : A \in \Gamma}{\Gamma \vdash x \leqslant x} \qquad \textsc{AS-VarTrans}\ \frac{x \leqslant e_1 : A \in \Gamma \qquad \Gamma \vdash e_1 \leqslant e_2}{\Gamma \vdash x \leqslant e_2} \qquad \textsc{AS-Top}\ \frac{}{\Gamma \vdash e \leqslant \top}$$

$$\textsc{AS-App}\ \frac{\Gamma \vdash e_1 \leqslant e_2}{\Gamma \vdash e_1\ A \leqslant e_2\ A} \qquad \textsc{AS-Abs}\ \frac{\Gamma, x \leqslant |e_3| : |A| \vdash e_1 \leqslant e_2}{\Gamma \vdash (\lambda x \leqslant e_3 : A.\ e_1) \leqslant (\lambda x \leqslant e_3 : A.\ e_2)}$$

$$\textsc{AS-Prod}\ \frac{\Gamma \vdash C \leqslant A \qquad \Gamma, x \leqslant |e| : |C| \vdash B \leqslant D}{\Gamma \vdash (\Pi x \leqslant e : A.\ B) \leqslant (\Pi x \leqslant e : C.\ D)} \qquad \textsc{AS-CastUp}\ \frac{\Gamma \vdash e_1 \leqslant e_2}{\Gamma \vdash \mathsf{cast}_{\uparrow}\ [A]\ e_1 \leqslant \mathsf{cast}_{\uparrow}\ [A]\ e_2}$$

$$\textsc{AS-CastDn}\ \frac{\Gamma \vdash e_1 \leqslant e_2}{\Gamma \vdash \mathsf{cast}_{\downarrow}\ e_1 \leqslant \mathsf{cast}_{\downarrow}\ e_2}$$

Figure 5.7. Algorithmic subtyping

5.4.1 Bidirectional Type Checking

We extend the syntax of $\lambda I_{\leqslant}$ with annotations, denoted by $(e : A)$ (parentheses are required). We use $|e|$ to denote the erasure of all annotation from a term and $|\Gamma|$ for the erasure of a context (see Figure 5.6). The algorithmic subtyping judgment is denoted by $\Gamma \vdash e_1 \leqslant e_2$ in Figure 5.7. It is developed by removing the typing part of unified subtyping rules in Figure 5.4.

The algorithmic typing judgment has two directions: the checking judgment $\Gamma \vdash e \Leftarrow A$ and the synthesis judgment $\Gamma \vdash e \Rightarrow A$, as shown in Figure 5.8. For brevity, we omit the context well-formedness in all algorithmic typing judgments. Typing rules are developed by following the typing part of unified subtyping. Most syntactic forms are typed by the synthesis judgment, including functions and function types since both binders are annotated. Two syntactic forms that are not annotated require the checking judgment, namely the top type ($\top$) and $\mathsf{cast}_{\downarrow}$ term. The subsumption rule from the unified subtyping is adapted to the checking direction.

$\boxed{\Gamma \vdash e \Rightarrow A}$ *(Synthesis)*

$$\textsc{AT-Ax}\ \frac{}{\Gamma \vdash \star \Rightarrow \star} \qquad \textsc{AT-Var}\ \frac{x \leqslant e : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{AT-Abs}\ \frac{\Gamma \vdash e_1 \Leftarrow |A| \quad \Gamma \vdash A \Leftarrow \star \quad \Gamma, x \leqslant |e_1| : |A| \vdash e_2 \Rightarrow B}{\Gamma \vdash \lambda x \leqslant e_1 : A.\ e_2 \Rightarrow \Pi x \leqslant |e_1| : |A|\ .\ B}$$

$$\textsc{AT-App}\ \frac{\Gamma \vdash e \Rightarrow \Pi x \leqslant e_3 : B.\ C \quad \Gamma \vdash A \Leftarrow B \quad \Gamma \vdash |A| \leqslant e_3}{\Gamma \vdash e\ A \Rightarrow C[x \mapsto |A|]} \qquad \textsc{AT-Prod}\ \frac{\Gamma \vdash e \Leftarrow |A| \quad \Gamma \vdash A \Leftarrow \star \quad \Gamma, x \leqslant |e| : |A| \vdash B \Leftarrow \star}{\Gamma \vdash \Pi x \leqslant e : A.\ B \Rightarrow \star}$$

$$\textsc{AT-CastUp}\ \frac{\Gamma \vdash B \Leftarrow \star \quad \Gamma \vdash e \Rightarrow A \quad |B| \hookrightarrow |A|}{\Gamma \vdash \mathsf{cast}_{\uparrow}\ [B]\ e \Rightarrow |B|} \qquad \textsc{AT-Anno}\ \frac{\Gamma \vdash e \Leftarrow A \quad \Gamma \vdash A \Leftarrow \star}{\Gamma \vdash (e : A) \Rightarrow |A|}$$

$\boxed{\Gamma \vdash e \Leftarrow A}$ *(Checking)*

$$\textsc{AT-Top}\ \frac{\Gamma \vdash A \Leftarrow \star}{\Gamma \vdash \top \Leftarrow A} \qquad \textsc{AT-CastDn}\ \frac{\Gamma \vdash B \Leftarrow \star \quad \Gamma \vdash e \Rightarrow A \quad |A| \hookrightarrow |B|}{\Gamma \vdash \mathsf{cast}_{\downarrow}\ e \Leftarrow B}$$

$$\textsc{AT-Sub}\ \frac{\Gamma \vdash e \Rightarrow A \quad \Gamma \vdash A \leqslant |B| \quad \Gamma \vdash B \Leftarrow \star}{\Gamma \vdash e \Leftarrow B} \qquad \textsc{AT-Chk}\ \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash e \Leftarrow A}$$

Figure 5.8. Bidirectional typing

We use erasure in the typing judgment to ensure there are no annotations in 1) the typing result and the context, 2) the terms being compared by the algorithmic subtyping judgment, and 3) the terms checked by the reduction relation ($\hookrightarrow$). However, if erasure is used in the typing result of a premise using synthesis, i.e., $\Gamma \vdash e \Rightarrow |A|$, the original form of A requires guessing. Referring to the original A in other premises renders the typing rule not algorithmic. Thus, we make sure there is no such form of synthesis in the premises of typing rules. Note that the typing rule is still algorithmic if the erased typing result appears in the conclusion of a synthesis rule or a premise using checking judgment.

5.4.2 Soundness and Completeness

We show that the algorithmic subtyping and typing are both sound and complete to the original unified subtyping. We use $\Gamma \vdash e \Leftrightarrow A$ to denote a judgment which can either be the checking judgment $\Gamma \vdash e \Leftarrow A$ or the synthesis judgment $\Gamma \vdash e \Rightarrow A$. The main theorems are stated as follows:

Theorem 5.4.1 (Soundness of Algorithm). *If* $\Gamma \vdash e_1 \Leftrightarrow A$, $\Gamma \vdash e_2 \Leftrightarrow A$ *and* $\Gamma \vdash e_1 \leqslant e_2$, *then* $|\Gamma| \vdash |e_1| \leqslant |e_2| : |A|$.

Theorem 5.4.2 (Completeness of Algorithm). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$, *then* $\Gamma \vdash e_1 \leqslant e_2$ *and there exists* e_1' *and* e_2' *such that* $\Gamma \vdash e_1' \Rightarrow A$ *and* $\Gamma \vdash e_2' \Rightarrow A$ *with* $|e_1'| = e_1$ *and* $|e_2'| = e_2$.

$\boxed{T^* = A}$ Mapping of Type

$$
\begin{aligned}
\top^* &= \top \\
X^* &= X \\
(T_1 \to T_2)^* &= \Pi x \leqslant \top : {T_1}^*.\ {T_2}^* \\
&\qquad (x \text{ Fresh}) \\
(\forall X \leqslant T_1.\ T_2)^* &= \Pi X \leqslant {T_1}^* : \star.\ {T_2}^*
\end{aligned}
$$

$\boxed{t^* = e}$ Mapping of Term

$$
\begin{aligned}
x^* &= x \\
(\lambda x : T.\ t)^* &= \lambda x \leqslant \top : T^*.\ t^* \\
(t_1\ t_2)^* &= {t_1}^*\ {t_2}^* \\
(\Lambda X \leqslant T.\ t)^* &= \lambda X \leqslant T^* : \star.\ t^* \\
(t[T])^* &= t^*\ T^*
\end{aligned}
$$

$\boxed{\Delta^* = \Gamma}$ Mapping of Context

$$
\begin{aligned}
\varnothing^* &= \varnothing \\
(\Delta, x : T)^* &= \Delta^*, x \leqslant \top : T^* \\
(\Delta, X \leqslant T)^* &= \Delta^*, X \leqslant T^* : \star
\end{aligned}
$$

Figure 5.9. Translation of System $F_{\leqslant}$

5.5 Subsumption of System $F_{\leqslant}$

$\lambda I_{\leqslant}$ is a generalization of System $F_{\leqslant}$ with dependent types. In this section, we show that $\lambda I_{\leqslant}$ can completely subsume the *Kernel Fun* variant [Cardelli and Wegner 1985] of System $F_{\leqslant}$. We first show the translation from System $F_{\leqslant}$ to $\lambda I_{\leqslant}$ and prove that the typing and subtyping judgments of System $F_{\leqslant}$ still hold in $\lambda I_{\leqslant}$ up to mapping. The full specification of System $F_{\leqslant}$ was presented in Section 2.4.2 (see Figure 2.3). The full proofs are available in Appendix A.2.

5.5.1 Translating System $F_{\leqslant}$ to $\lambda I_{\leqslant}$

We show the mapping (denoted by *) of types, terms and contexts from System $F_{\leqslant}$ to $\lambda I_{\leqslant}$ in Figure 5.9. We use the metavariable T for types, t for terms and Δ for contexts in System $F_{\leqslant}$. The arrow type is non-dependent and unbounded and therefore mapped to a top-bounded function type, similar to the treatment of syntactic sugar in Figure 5.2. The universal type is mapped to the dependent function type since X can appear in T_2. The bound T_1 is a proper type and mapped to ${T_1}^*$ with kind $\star$. The term and type abstraction, as well as term and type binding of the context, are treated similarly. Other mappings hold few surprises.

5.5.2 Subsumption of Typing and Subtyping

We prove that the mapped typing and subtyping relations still hold in $\lambda I_{\leqslant}$. The type system of System $F_{\leqslant}$ we used here is the *algorithmic* [Curien and Ghelli 1992] and *Kernel Fun* variant [Cardelli and Wegner 1985]. We first show the well-formedness of types and contexts still hold after mapping:

Lemma 5.5.1 (Mapping of Well-formedness ✎). *(1) If* $\Delta \vdash T$*, then* $\Delta^* \vdash T^* : \star$*; (2) If* $\vdash \Delta$*, then* $\vdash \Delta^*$.

The proof is by simultaneous induction on the derivation of well-formedness of types $\Delta \vdash T$ and contexts $\vdash \Delta$. Then we show the mapped subtyping and typing still hold:

Theorem 5.5.1 (Subsumption of Subtyping ✎). *If* $\Delta \vdash T_1 \leqslant T_2$*, then* $\Delta^* \vdash {T_1}^* \leqslant {T_2}^* : \star$.

Theorem 5.5.2 (Subsumption of Typing ✎). *If* $\Delta \vdash t : T$*, then* $\Delta^* \vdash t^* : T^*$.

The proof is straightforward by induction on the derivation of subtyping relation $\Delta \vdash T_1 \leqslant T_2$ and typing relation $\Delta \vdash t : T$, respectively. Note that the mapped typing relation $\Delta^* \vdash t^* : T^*$ is syntactic sugar of unified subtyping relation, i.e., $\Delta^* \vdash t^* \leqslant t^* : T^*$ (see Figure 5.2).

5.6 Discussion

In this section, we discuss alternative designs for $\lambda I_{\leqslant}$ and justify their trade-offs to the current design.

Recursion and Recursive Types. The current syntax of $\lambda I_{\leqslant}$ does not contain any form of recursion. Adding recursion and recursive types is easy by simply following the treatment of recursion in PITS. We have an alternative formulation of our system (including full proofs) with those features. However subtyping recursive types reveals an interesting problem. The typical Amber rule [Cardelli 1986a], or even the following restricted invariant rule

$$\frac{\Gamma, x \leqslant \top : A \vdash e_1 \leqslant e_2 : A \qquad \Gamma \vdash A : \star}{\Gamma \vdash (\mu x : A.\ e_1) \leqslant (\mu x : A.\ e_2) : A} \text{ S-MuI}$$

does not work well with $\lambda I_{\leqslant}$. Here $\mu x : A.\ e_1$ is a recursive type with the recursive binder x that can appear in the body e_1. The rule requires the types of recursive binders to be the same. We add a new reduction rule to unfold a recursive type: $\mu x : A.\ e \hookrightarrow e[x \mapsto \mu x : A.\ e]$. In order to keep type soundness, we need to ensure subtype preservation (Lemma 5.3.14) still holds. If $f = \lambda y : \star.\ y$ is an identity type operator with type $\star \to \star$, consider

$$\mu x : \star.\ f\ x \leqslant \mu x : \star.\ \top\ x$$

This relation holds by the rule S-MuI because we have $f \leqslant \top : \star \to \star$ by S-Top and then $x : \star \vdash f\ x \leqslant \top\ x : \star$ by S-App. Subtype preservation requires that the subtyping relation still holds with both sides reduced by one step:

$$f\ (\mu x : \star.\ f\ x) \leqslant \top\ (\mu x : \star.\ \top\ x) \tag{5.1}$$

However, (5.1) does not hold because the pointwise subtyping rule S-App requires arguments of two applications should be the same. Thus, types are not preserved using the invariant rule for subtyping recursive types. This issue appears to be common to most systems with *higher-order subtyping* [Pierce and Steffen 1997; Aspinall and Compagnoni 1996; Zwanenburg 1999], as it arises from the interaction between the rules for recursive types and rules that use pointwise subtyping.

To solve this issue, we either change the S-App rule to be polarized [Steffen 1998], or only allow subtyping two identical recursive types. The former approach is interesting, but requires a major modification to the system. We leave that approach for future work (see Section 8.2). The latter approach is relatively simple by using the following rule:

$$\frac{\Gamma, x \leqslant \top : A \vdash e : A \qquad \Gamma \vdash A : \star}{\Gamma \vdash (\mu x : A.\ e) \leqslant (\mu x : A.\ e) : A} \text{ S-Mu}$$

Due to the unified syntax, $\mu x : A.\ e$ can serve as both the term-level fixpoint and recursive type. Though full subtyping of recursive types is not possible in $\lambda I_{\leqslant}$ currently, we are still able to

introduce general recursion and recursive types to the system with S-Mu. This is precisely the approach used in our alternative formulation.

Operational Semantics. $\lambda I_{\leqslant}$ uses the same call-by-name (CBN) operational semantics that call-by-name PITS uses (see Section 4.1). However most OO languages use call-by-value (CBV). CBV semantics is more complicated because of the existence of dependent types and explicit casts in $\lambda I_{\leqslant}$. We also treat $\mathsf{cast}_{\uparrow}\ [A]\ e$ as a value (see Section 5.2.2), which follows the standard call-by-name semantics of iso-recursive types [Harper 2013]. Such design makes the $\mathsf{cast}_{\uparrow}$ operator *computationally relevant* (see Section 4.3.1). Alternatively, we can take the approach from call-by-value PITS (see Section 4.2), which treats $\mathsf{cast}_{\uparrow}\ [A]\ v$ as a value and adds a reduction rule to further reduce the inner term of $\mathsf{cast}_{\uparrow}$. However, the alternative semantics of $\mathsf{cast}_{\uparrow}$ leads to more complex reduction rules and metatheory. The cast canceling rule R-CastElim (See Figure 5.3) now needs to check if the inner term of $\mathsf{cast}_{\uparrow}$ is a value, which requires some non-trivial changes to current proofs of the metatheory. We will later discuss the use of call-by-value casts for typing strong dependent sums in the next chapter (see Section 6.1.2).

Top Types. For top types, we can alternatively treat only $\top$ as a value but not $\top\ e_1\ \ldots\ e_n$, which is an inert term (see Figure 5.2). In such design additional reduction rules similar to the $\beta\top$-reduction rules of System $F^{\omega}_{\leqslant}$ [Pierce and Steffen 1997] are needed to further reduce "stuck" terms to values, i.e., $\top\ e \hookrightarrow \top$. However, the approach of using $\beta\top$-reduction needs to define reduction rules for each form of stuck terms, e.g., $\top\ e$ and $\mathsf{cast}_{\downarrow}\ \top$, while the definition of inert terms deals with stuck terms in a more uniform way.

Weak vs Full Casts. Cast operators in $\lambda I_{\leqslant}$ use the same weak-head reduction for type-level computation. As mentioned in Section 5.1.4, certain type conversions cannot be performed by weak-head reduction/expansion if they require reduction at non-head position, e.g., converting $\mathit{Vec}\ (1+1)$ to $\mathit{Vec}\ 2$. To address this limitation we can use an alternative design from the full PITS (see Section 4.3). Full PITS uses full parallel reduction is used in cast operators, which allows reduction at any position of a term. However the metatheory of full PITS is significantly more complicated than call-by-name/value PITS. Since weak-head reduction was simpler and sufficient for our purposes (to model object encodings) we opted for that variant. It would be interesting to study the full-cast variant of PITS with subtyping as well in future work.

Full Contravariance of Function Types. As mentioned in Section 5.2.3, the unified subtyping rule of function types is *partially* contravariant in the sense that bounds of function types are identical, which follows the treatment of universal types in the Kernel Fun variant [Cardelli and Wegner 1985] of System $F_{\leqslant}$. An alternative is to follow the *full* System $F_{\leqslant}$ that allows bounds to be contravariant:

$$\frac{\Gamma \vdash A' \leqslant A : \star \qquad \Gamma \vdash e' \leqslant e : A \qquad \Gamma, x \leqslant e' : A' \vdash B \leqslant B' : \star \qquad \Gamma \vdash e : A \qquad \Gamma \vdash e' : A' \qquad \Gamma \vdash A : \star \qquad \Gamma, x \leqslant e : A \vdash B : \star}{\Gamma \vdash (\Pi x \leqslant e : A.\ B) \leqslant (\Pi x \leqslant e' : A'.\ B') : \star}\ \text{S-ProdF}$$

We formulated an alternative system with such full contravariant rule and proved all lemmas in Section 5.3 still hold. The corresponding Coq formalization can be found with the companion materials of this thesis available online (See Section 1.5). However, full System $F_{\leqslant}$ is proved to be

undecidable [Pierce 1992]. With contravariance of bounds, $\lambda I_{\leqslant}$ using rule S-ProdF can subsume full System $F_{\leqslant}$, rendering the system undecidable. Though we have not proved the decidability of $\lambda I_{\leqslant}$ yet, we adopt the Kernel-Fun rule in $\lambda I_{\leqslant}$ and can at least rule out the undecidability caused by the full contravariance.

CHAPTER 6

Iso-Types with Strong Dependent Sums

In this chapter, we present the λI_{Σ} calculus. The motivation of proposing λI_{Σ} is to model OOP structures with abstract type members using dependent types and strong dependent sums. The λI_{Σ} calculus is a variant of $\lambda I_{\leqslant}$ and enables the combination of strong sums and dependent types. It is based on the ideas of iso-types and unified subtyping. λI_{Σ} employs the PTS-style unified syntax and contains the single sort $\star$ and the "type-in-type" axiom as in $\lambda I_{\leqslant}$. Moreover, λI_{Σ} supports impredicative polymorphism with strong sums, which usually cannot be supported simultaneously by previous work [Harper and Mitchell 1993; Stump 2017; Bowman et al. 2017]. Impredicativity and strong sums together lead to logical inconsistency [Coquand 1986; Hook and Howe 1986] and breaks strong normalization. Moreover, some dependently typed calculi with subtyping, such as $\lambda P_{\leqslant}$ [Aspinall and Compagnoni 1996] and $\lambda\Pi_{\leqslant}$ [Chen 1998], rely on strong normalization to prove many desirable properties, such as transitivity and subject reduction, which will be lost if both impredicativity and strong sums are allowed. In contrast λI_{Σ} uses the iso-type approach that decouples proofs from strong normalization. It enjoys the same desirable properties as $\lambda I_{\leqslant}$, e.g., transitivity of subtyping and type-safety, which can be proved in the absence of strong normalization.

Iso-types are used in λI_{Σ} to type destructors of strong sums. We call this form of strong sums *iso-strong sums*. Unlike the call-by-name casts in $\lambda I_{\leqslant}$, call-by-value casts are used in λI_{Σ} for type conversions of strong sums. The typing results of strong destructors (i.e. the second projection and strong opening) are *intermediate* type-level applications instead of standard direct substitutions. This makes it possible to solely use call-by-value casts for the required type-level computation instead of full casts. Using call-by-value casts also makes it easy to combine unified subtyping and avoids the complex metatheory of full casts.

Notice that for simplicity reasons, several features are dropped in λI_{Σ} so as to focus on the development of dependent sums, such as generalized top type and bounded quantification. Nevertheless, λI_{Σ} is still expressive enough to encode Scala-like type members and traits. Finally, for demonstration purposes, we create a lightweight surface language **Sig** built on top of λI_{Σ}. **Sig** supports Scala-like type members, path-dependent types and traits by a type-directed elaboration that produces type-sound λI_{Σ} terms.

Table 6.1. Properties of dependently typed calculi with beta equality

Features	Example Calculi	Properties
Π+Impredicativity	λC	Subject Reduction **does not depend on** Strong Normalization.
Π+Σ+Impredicativity	λC with strong sums	**No** Strong Normalization or Logical Consistency, but Subject Reduction **holds**.
Π+Subtyping	$\lambda P_{\leqslant}$ and $\lambda\Pi_{\leqslant}$	Transitivity & Subject Reduction **depend on** Strong Normalization.
Π+Σ+Subtyping+ Impredicativity	No known calculus	Strong Normalization, Transitivity & Subject Reduction **may not hold**.

6.1 Overview

In this section, we informally present features of λI_{Σ}. The motivation of developing the λI_{Σ} calculus is to support various features of strong dependent sums, which are useful to model many language constructs, including type members and traits in Scala [Odersky et al. 2004], as well as various features of module systems in ML [MacQueen 1986]. However, it is non-trivial to combine strong sums, dependent types and subtyping in a single system. We briefly discuss such problem and present our solution in λI_{Σ}, i.e. *iso-strong sums* that utilize call-by-value casts without the need of full casts. We show how to encode modular structures with strong sums by examples written in **Sig**, a lightweight surface language built on top of λI_{Σ}. The formal presentation of λI_{Σ} and **Sig** will be in Sections 6.2 and 6.4, respectively.

6.1.1 The Trouble with Impredicativity and Strong Sums

Strong sum types (Sigma-types) bring extra expressiveness over dependent function types (Pi-types). However, it is non-trivial to combine strong sums with dependent type systems and keep desirable properties. For example, the calculus of constructions (λC) with strong sums is *logically inconsistent* [Coquand 1986; Hook and Howe 1986], since the inclusion of strong sums leads to the subsumption of the "type-in-type" axiom. The subsumption of "type-in-type" is enabled by the combination of strong sums and *impredicativity*. A system is impredicative if it contains an impredicative sort s where certain types of sort s have quantifiers of the sort s itself [Adams 2008]. For example, the sort $\star$ in λC is an impredicative sort since $\Pi x : \star.\ x$ has type $\star$. As a result, strong normalization does not hold in such system and some other properties may also be lost, e.g., the decidability of type checking, which is entangled with the normalization property in λC (also other PTS) [van Benthem Jutting 1993].

The Problem of Adding Subtyping. Subtyping is a desirable language feature to support, which is crucial for modeling parametric polymorphism in OOP or information hiding in ML modules [Dreyer 2005; Leroy 1994]. However, combining subtyping and dependent types is already difficult (see Section 5.1.2), resulting in a more complex metatheory. In certain calculi that support both dependent types and subtyping, several important metatheory results depend on strong normalization. For example, in $\lambda P_{\leqslant}$ [Aspinall and Compagnoni 1996] and $\lambda\Pi_{\leqslant}$ [Castagna and Chen 2001; Chen 1998], both subject reduction and transitivity of subtyping rely on strong normalization. In contrast, λC and also PTS do not require strong normalization to prove subject reduction. We summarize the properties of previously mentioned calculi in Table 6.1 where Π

stands for dependent function types and Σ stands for strong sum types. The situation becomes more complex when further considering to combine subtyping and strong sums. If we add strong sums to calculi such as $\lambda P_{\leqslant}$ and $\lambda\Pi_{\leqslant}$, allowing impredicativity will break strong normalization and eventually break subject reduction and transitivity, as indicated by the last row of Table 6.1. Thus, it becomes rather difficult to simultaneously allow impredicativity, subtyping and strong sums and keep all desirable properties in such calculi.

Previous Solution: Abandoning Impredicativity. One approach to supporting strong sums without breaking logical consistency and strong normalization is to abandon impredicativity. For example, Harper and Mitchell [1993] proposed XML, an extension of Standard ML with strong sums, which supports only predicative polymorphism by stratifying types into different universes. Dropping impredicativity is reasonable in the context of an ML-like language, since predicativity is crucial for other language features, e.g., let-polymorphism and type inference. However, such restriction still significantly limits the expressiveness of the language, making it unable to fully subsume System F or $F_{\leqslant}$ which both supports impredicative polymorphism.

6.1.2 Iso-Strong Sums: Typing Strong Sums with Iso-Types

Unlike the previous work, λI_{Σ} features *iso-strong sums*, which employs the iso-type approach and uses explicit *call-by-value* cast operators for type-level computation with strong sums. By using iso-types, the metatheory proofs are decoupled from strong normalization (see Sections 3.1.3 and 5.1.4). In λI_{Σ}, we can prove transitivity of (unified) subtyping and subject reduction without requiring normalization (see Section 6.3). Thus, impredicative polymorphism, (iso-type style) strong sums and (unified) subtyping can coexist in λI_{Σ} without breaking desirable properties. In the rest of this subsection, we show how iso-strong sums are typed in λI_{Σ} through call-by-value casts.

Standard Typing of Second Projection. The typing of strong dependent sums involves type-level computation. Specifically, the type of the second projection of a term depends on its first projection. It is the key complication over weak sums which do not support second projection but only weak existential opening. Given a term e that has a Sigma-type, the standard typing of its second projection $e.2$ is shown as follows:

$$\frac{\Gamma \vdash e : \Sigma x : A.\ B}{\Gamma \vdash e.2 : B[x \mapsto e.1]}$$

where the type of $e.2$ is a substitution and depends on $e.1$. Consider an expression e which is exactly a dependent sum, i.e.,

$$\begin{aligned} e &= \langle e_1, e_2 \text{ as } \Sigma x : A.\ B \rangle \\ e_1 &: A \\ e_2 &: B[x \mapsto e_1] \end{aligned}$$

where the type of the second component depends on the first. Noting that the first projection $e.1$ can be reduced to the first component e_1, we need type-level computation to convert $e.1$ into e_1:

$$\begin{array}{ll} & \langle e_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.1 \hookrightarrow e_1 \\ e.2 & : B[x \mapsto \langle e_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.1] \\ e.2 & : B[x \mapsto e_1] \end{array}$$

After conversion, the type of $e.2$ matches the type of e_2 and the reduction $\langle e_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.2 \hookrightarrow e_2$ preserves the type.

Second Projections and Alpha Equality. In λI_Σ, there is no implicit type conversion and we need to add cast operators explicitly for such conversion:

$$\begin{array}{ll} e.2 & : B[x \mapsto \langle e_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.1] \\ \mathsf{cast}_{\Downarrow}\ (e.2) & : B[x \mapsto e_1] \end{array}$$

The full $\mathsf{cast}_{\Downarrow}$ operator[1] from the full PITS (see Section 4.3) is required here, which utilizes parallel reduction. The call-by-name/value reduction is not congruent and not strong enough to convert such types inside the substitution. For example, consider $e = \langle Int, (\lambda y : Int.\ y) \text{ as } \Sigma x : \star.\ (x \to x)\rangle$. Then we have $e.1 \hookrightarrow Int$ and

$$\begin{array}{ll} e & : \Sigma x : \star.\ (x \to x) \\ e.2 & : (e.1) \to (e.1) \\ \mathsf{cast}_{\Downarrow}\ (e.2) & : Int \to Int \end{array}$$

Notice that the type of $e.2$ is a Pi-type (i.e. $\Pi y : (e.1).\ (e.1)$) which is not reducible by call-by-name/value reduction. However, as discussed in Section 4.3, using full casts significantly complicates the metatheory and makes it difficult to support subtyping. With full casts, one relies on a separate system that erases all casts for proving type soundness. The technique of unified subtyping that uses iso-types does not apply in such erased system, making it hard to prove relevant properties, such as transitivity and subject reduction. We still prefer call-by-name/value reduction for explicit casts, which is compatible with the existing method of unified subtyping.

Typing Second Projection with CBV Casts. Alternatively to full casts, we can slightly weaken the type of second projection and make it convertible with *call-by-value* casts. The approach is to "take one step back" by changing the type substitution back into an application form as follows[2]:

$$\frac{\Gamma \vdash e : \Sigma x : A.\ B}{\Gamma \vdash e.2 : (\lambda x : A.\ B)\,(e.1)}$$

The original standard type "$B[x \mapsto e.1]$" can be imagined as a reduced term of the current type. The new typing result of $e.2$ can be seen as an *intermediate* step where $e.1$ is not yet substituted into B and remains as an argument. Thus, we can use call-by-value reduction to further reduce $e.1$

[1] For brevity, we leave out the annotation of $\mathsf{cast}_{\Downarrow}$.

[2] The rule presented here is for demonstrating the idea of typing and is a bit different from the one used in λI_Σ. It does not employ the value restriction, i.e., requiring e to be a value v. We will later discuss the value restriction of λI_Σ in Section 6.2.

until it becomes a value. For example, suppose that e is a dependent sum whose first component is a value:

$$\begin{aligned} e &= \langle v_1, e_2 \text{ as } \Sigma x : A.\ B\rangle \\ e.2 &: (\lambda x : A.\ B)\,(\langle v_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.1) \end{aligned}$$

Note that a dependent sum is already a value. We can use two consecutive call-by-value $\mathsf{cast}_{\downarrow}$ operators to obtain the desired type:

$$\begin{aligned} e.2 &: (\lambda x : A.\ B)\,(\langle v_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.1) \\ \mathsf{cast}_{\downarrow}\,(e.2) &: (\lambda x : A.\ B)\,v_1 \\ \mathsf{cast}^2_{\downarrow}\,(e.2) &: B[x \mapsto v_1] \end{aligned}$$

Specifically, we use one cast for reducing the first projection and the other cast for beta reduction, i.e.,

$$\begin{aligned} \langle v_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.1 &\hookrightarrow v_1 \\ (\lambda x : A.\ B)\,v_1 &\hookrightarrow B[x \mapsto v_1] \end{aligned}$$

The type of $\mathsf{cast}^2_{\downarrow}\,(e.2)$ matches the type of e_2. In the reduction rule for $\langle v_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.2$, we need to correspondingly add two extra $\mathsf{cast}_{\uparrow}$ operators for the reduced term needs to balance the type:

$$\langle e_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.2 \hookrightarrow \mathsf{cast}^2_{\uparrow}\ e_2$$

Then the whole reduction process preserves the types:

$$\mathsf{cast}^2_{\downarrow}\,(\langle v_1, e_2 \text{ as } \Sigma x : A.\ B\rangle.2) \hookrightarrow \mathsf{cast}^2_{\downarrow}\,(\mathsf{cast}^2_{\uparrow}\ e_2) \hookrightarrow \ldots \hookrightarrow e_2$$

Thus by weakening the substitution form into a type-level application, we can now convert the type of second projection using non-congruent call-by-value reduction without the need of full parallel reduction. The trade-off is that we need more type conversion steps with CBV cast than full casts, e.g., two call-by-value $\mathsf{cast}_{\downarrow}$ operators but only one $\mathsf{cast}_{\Downarrow}$ in the example above. Nevertheless, CBV casts are still able to perform necessary type conversions for strong sums. Also, subtyping in λI_{Σ} can be supported by applying the technique of unified subtyping which is compatible with CBV casts.

6.1.3 Example: Type Members and Traits

In Sections 1.3.3 and 2.2.3, we showed that Scala-like traits with type members can be encoded using dependent sums. Recall the example of an abstract integer set trait. We can write the same example in **Sig**, a simple surface language over λI_{Σ}, as follows:

```
Set = trait {
    type T: Type;
    val empty: T;
    val member: Int -> T -> Bool;
    val insert: Int -> T -> T;
}
```

We use monospaced font for **Sig** programs. We can translate `Set` to a Sigma-type in λI_{Σ} (assuming that we have records and primitive types such as integers and booleans):

$$Set = \Sigma T : \star.\{$$

$$
\begin{aligned}
&\quad empty : T,\\
&\quad member : Int \to T \to Bool,\\
&\quad insert : Int \to T \to T,\\
&\}
\end{aligned}
$$

The type member T corresponds to the type binder of the Sigma-type. The methods of the trait are encoded as record members of the record type: `empty` returns an empty set; `member` checks if an element is in the set; and `insert` adds an element into the set. We can also write the generic function `f` from Section 2.2.3 in **Sig**:

```
f = λ(s:Set) => s.member 3 (s.insert 3 s.empty)
```

which is encoded in λI_{Σ} by the **open** operator, a stronger version of the **unpack** operator of weak sums:

$$f = \lambda s : Set.\ \mathbf{open}\ s\ \mathbf{as}\ \langle T, s' \rangle\ \mathbf{in}\ s'.member\ 3\ (s'.insert\ 3\ s'.empty)$$

Limitations of Weak Sums. The examples by far can be encoded using only weak sums. However, weak sums are not sufficient to encode all type member features in Scala, such as path-dependent types. The function f is encodable by weak sums because the type of the whole function body is $Bool$ and does not contain the abstract type member T. Recall the same function `g` from Section 2.2.3, a simple function that returns an empty set:

```
g = λ(s:Set) => s.empty
```

Note that the type of the function body now refers to the abstract type member. In Scala, the type of `s.empty` is `s.T`, which is a *path-dependent type*. We cannot translate `s.empty` with weak sums, such as "$\mathbf{unpack}\ s\ \mathbf{as}\ [T, s']\ \mathbf{in}\ s'.empty$", because $s'.empty$ has type T which is out of the scope of the **unpack**-term.

Encoding with Iso-Strong Sums. In λI_{Σ}, we can encode `g` with **open**:

$$g = \lambda s : Set.\ \mathbf{open}\ s\ \mathbf{as}\ \langle T, s' \rangle\ \mathbf{in}\ s'.empty$$

Unlike **unpack**, the **open** operator permits the body to refer to the abstract type member T. In λI_{Σ}, the type of g is as follows:

$$g : \Pi s : Set.\ (\lambda T : \star.\ T)\ (s.1)$$

where $s.1$ is the projection of the first component of s. Similarly to the treatment of typing second projection (see Section 6.1.2), the type of strong **open** is a *type-level application*: $(\lambda T : \star.\ T)\ (s.1)$. Essentially, we know that the type of the **open**-term should be the same as the actual type member of s, i.e. $s.1$, the first component of s. But s is abstract and we do not know what the actual type member is yet. The type-level application can be viewed as an *intermediate step*, which is waiting for the actual implementation of s. Once s is given, we can continue the type-level computation with *explicit casts* to obtain the final type. Consider s is instantiated by `ListSet`, a set implementation using a list. We borrow the syntax of Haskell's list type and functions in the following **Sig** pseudo-code:

```
ListSet = obj {
    type T = [Int];
    val empty = [];
    val member = elem;
    val insert = (:);
}
```

which is translated to a dependent sum in λI_{Σ}:

$$ListSet = \langle [Int], \{\, empty = [\,], member = elem, insert = (:)\,\} \text{ as } Set \rangle$$

Two components of the dependent sum are wrapped by angle brackets and separated by a comma. The type annotation Set is put after the keyword as. Then the first projection of $ListSet$ is its first component $[Int]$, i.e., $ListSet.1 \hookrightarrow [Int]$. We can eliminate the type-level application as follows:

$$\begin{array}{ll} g\ ListSet & : (\lambda T : \star.\ T)\ (ListSet.1) \\ \mathsf{cast}_{\downarrow}\ (g\ ListSet) & : (\lambda T : \star.\ T)\ [Int] \\ \mathsf{cast}^2_{\downarrow}\ (g\ ListSet) & : [Int] \end{array}$$

which needs two consecutive $\mathsf{cast}_{\downarrow}$ operators to obtain the target type: one for reducing the first projection, and the other one for reducing the application.

We emphasize three points from the translation of g:

- First, the encoding of g relies on the exclusive features of strong sums in λI_{Σ}, thus cannot be encoded with weak sums in $\lambda I_{\leqslant}$.
- Second, the reduction relation used in $\mathsf{cast}_{\downarrow}$ is *call-by-value*, which reduces the argument into a value before beta-reduction. Using call-by-value casts is more appropriate here because call-by-name reduction cannot reduce the argument and does not meet the purpose. The full cast with parallel reduction is over powerful and complicates the metatheory (see Section 4.3), which makes it difficult to combine with other features, such as subtyping.
- Third, we use a strong **open** operator here instead of the second projection (see Section 2.2.3) due to the typing of iso-strong sums. Otherwise, the typing result of the second projection of a iso-strong sum is a type-level application but not directly a record type. This forbids us to do further record projection to obtain the field $empty$. We will later discuss this issue in Section 6.2.3.

6.1.4 ML Module Systems and Strong Sums

The ML family of languages, such as Standard ML [Milner et al. 1990] and OCaml [Leroy et al. 2018], usually contains an advanced module system. The basic constructs are *signatures*, *structures* and *functors*. A structure is a module that consists of types and values. A signature is the type of a structure, containing the specification for each component of the structure. A functor is a function from structures to structures. Many languages have similar constructs to ML modules. For example in Scala (also **Sig**), traits correspond to signatures and objects correspond to structures. The **Sig** examples in Section 6.1.3, i.e., `Set` trait and `ListSet` object, can be easily adapted to constructs of the module system in OCaml:

```
module type Set = sig
  type t
  val empty: t
  val member: int -> t -> bool
  val insert: int -> t -> t
end
```

```
module ListSet = struct
  type t = int list
  let empty = []
  let member = List.mem
  let insert x l = x :: l
end
```

Functions of objects correspond to functors. For example, the function `g` can be modeled as a functor `G` that takes a `Set` parameter and returns a new structure wrapping the actual function `g`:

```
module G(S:Set) = struct
  let g = S.empty
end
```

The function application `g ListSet` can be simulated by a functor application `G(ListSet)`:

```
let module M = G(ListSet) in M.g
```

First-class Modules. In a certain sense, we can treat **Sig** as a very simple ML-like module system, though many features are missing in **Sig**, such as transparent types [Lillibridge 1997; Leroy 1994; Harper and Lillibridge 1994; Harper and Mitchell 1993]. It is not a coincidence since ML modules can also be modeled by strong dependent sums [MacQueen 1986]. Specifically, "modules" (objects indeed) are *first-class* values in **Sig**. Usually, ML modules are *second-class* and separated from terms and types. The core language of ML cannot freely manipulate modules, e.g., choosing modules at run-time:

```
module FastSet = if size > 20 then HashSet else ListSet
```

In **Sig**, there is no such restriction and modules/objects can be even more expressive due to dependent types. It is possible to define a trait such that the type of trait members can depend on values, for example:

```
BitMap = trait {
    type Dim: {m:Int, n:Int};
    val pixels: Array<Dim.m, Dim.n>;
}
```

The dimension type `Dim` is a record with two integers, which represents maximum rows and columns of a bitmap. The member `pixels` is a two-dimensional bit array that stores the data. The array type is a dependent type: it depends on the numbers `m` and `n` given by `Dim`. The `BitMap` trait encapsulates the information of dimensions. All `BitMap` objects have the same type, even if their dimensions are different. This makes it easier to define generic functions on the trait than the dependently-typed array type, i.e., `Array<m, n>`, since arrays have different types if their dimensions are not the same.

6.2 The λI_{Σ} Calculus

In this section, we formally introduce the λI_{Σ} calculus, a variant to $\lambda I_{\leqslant}$ with *iso-strong sums*. λI_{Σ} keeps core features of $\lambda I_{\leqslant}$, including iso-types, unified syntax and unified subtyping. To focus on the support of dependent sums and simplify the metatheory, several features are dropped, such as bounded quantification and generalized top type. λI_{Σ} also features a call-by-value semantics

for type casts, which enables the typing of second projection on dependent sums. The value restriction is employed similarly to call-by-value PITS. We present the syntax, dynamic and static semantics of λI_{Σ} in the rest of this section.

6.2.1 Syntax

The syntax of λI_{Σ} is shown in Figure 6.1. Similarly to $\lambda I_{\leqslant}$, there is no distinction between terms and types. The novelties over $\lambda I_{\leqslant}$ are language constructs for dependent sums. We follow the similar convention that metavariables are lowercase for terms and uppercase for types.

Dependent Sums. The dependent sum type, i.e., Sigma-type, is denoted by $\Sigma x : A.\ B$. Dependent sums are denoted by $\langle e_1, e_2 \ \mathsf{as}\ \Sigma x : A.\ B\rangle$ with the dependent sum type annotated after the as keyword. The type annotation is provided to simplify typing and reduction rules. As mentioned in Section 5.1.5, only *weak* dependent sums can be encoded. To obtain native support for *strong* dependent sums, we treat both constructs as primitives in λI_{Σ}.

Projection and Opening. λI_{Σ} supports two kinds of operations on dependent sums: projection and opening. Dependent sum types can be seen as a generalization to both product type $A \times B$ and existential type $\exists x.B$. The projection operator follows the dot syntax commonly used for pairs. The first (second) projection $e.1$ ($e.2$) extracts the first (second) component of the dependent sum e. The unrestricted second projection $e.2$ can only be supported with strong dependent sums. The opening expression $\mathbf{open}\ e_1\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ e_2$ follows the semantics of existential opening, which is similar to a pattern-matching binding. Expression e_1 is destructed into first and second components, which are bound to pattern variables x and y, respectively. The pattern variables can be used in another expression e_2. The opening operation is *strong* in the sense that x can show in the type of e_2, which is prohibited by the scoping restriction of weak existential opening. Both projection and opening are destructors of dependent sums and their usage may overlap. For example, we will show the second projection can actually be expressed by strong opening (see Section 6.2.3). We keep an individual form of the second projection for better presenting the system. However, we cannot leave out the form of the first projection, which is used in the typing result of second projection and opening (see Figure 6.3).

Other Changes to $\lambda I_{\leqslant}$. There are several simplifications in λI_{Σ} for focusing on discussion of dependent sums. We drop bounded quantification, which is an orthogonal feature to dependent sums. This significantly simplifies the syntax since the binder does not need to contain subtyping constraints. The definition of the context Γ only contains type bindings $x : A$. Without the demand to unify two forms of binders in $\lambda I_{\leqslant}$ (i.e. $\lambda x : A.\ e$ and $\lambda x \leqslant e' : A.\ e$), we also drop the generalized top type that allows $\top$ to have any type. In λI_{Σ}, $\top$ is only a *type* and has kind $\star$. The ordinary top type also makes type-safety proofs easier (see Section 6.3.3). Finally, we drop the annotation in the $\mathsf{cast}_{\uparrow}$ operator for simplifying reduction rules (see Section 6.2.2). Though type checking an unannotated $\mathsf{cast}_{\uparrow}$ expression is not algorithmic, the unified subtyping judgment is declarative anyway due to the subsumption rule (see Section 6.2.3).

$$
\begin{array}{llcl}
\text{Expressions} & e, A, B & ::= & x \mid \star \mid \top \mid e_1\ e_2 \mid \mathsf{cast}_{\uparrow}\ e \mid \mathsf{cast}_{\downarrow}\ e \\
& & \mid & \lambda x : A.\ e \mid \Pi x : A.\ B \\
& & \mid & \Sigma x : A.\ B \mid \langle e_1, e_2\ \mathsf{as}\ \Sigma x : A.\ B\rangle \\
& & \mid & e.1 \mid e.2 \mid \mathbf{open}\ e_1\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ e_2 \\
\text{Contexts} & \Gamma & ::= & \varnothing \mid \Gamma, x : A \\
\text{Values} & v & ::= & \star \mid x \mid \top \mid \lambda x : A.\ e \mid \Pi x : A.\ B \\
& & \mid & \mathsf{cast}_{\uparrow}\ e \mid \Sigma x : A.\ B \mid \langle v, e\ \mathsf{as}\ \Sigma x : A.\ B\rangle \\
\text{Syntactic Sugar} & A \to B & \triangleq & \Pi x : A.\ B \quad \text{where } x \notin \mathsf{FV}(B)
\end{array}
$$

Figure 6.1. Syntax of λI_{Σ}

$\boxed{e_1 \hookrightarrow e_2}$ *(Partially Call-by-Value Reduction)*

$$\text{RP-Beta}\ \frac{}{(\lambda x : A.\ e)\ v \hookrightarrow e[x \mapsto v]} \qquad \text{RP-AppL}\ \frac{e \hookrightarrow e'}{e\ v \hookrightarrow e'\ v} \qquad \text{RP-AppR}\ \frac{e_2 \hookrightarrow e_2'}{e_1\ e_2 \hookrightarrow e_1\ e_2'} \qquad \text{RP-CastDn}\ \frac{e \hookrightarrow e'}{\mathsf{cast}_{\downarrow}\ e \hookrightarrow \mathsf{cast}_{\downarrow}\ e'}$$

$$\text{RP-CastElim}\ \frac{}{\mathsf{cast}_{\downarrow}\ (\mathsf{cast}_{\uparrow}\ e) \hookrightarrow e} \qquad \text{RP-Fst}\ \frac{e \hookrightarrow e'}{e.1 \hookrightarrow e'.1} \qquad \text{RP-Proj1}\ \frac{}{\langle v, e\ \mathsf{as}\ \Sigma x : A.\ B\rangle.1 \hookrightarrow v}$$

$$\text{RP-Proj2}\ \frac{}{\langle v, e\ \mathsf{as}\ \Sigma x : A.\ B\rangle.2 \hookrightarrow \mathsf{cast}_{\uparrow}\ (\mathsf{cast}_{\uparrow}\ e)}$$

$$\text{RP-Open}\ \frac{}{\mathbf{open}\ \langle v, e_1\ \mathsf{as}\ \Sigma x : A.\ B\rangle\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ e \hookrightarrow \mathsf{cast}_{\uparrow}\ (\mathsf{cast}_{\uparrow}\ (((\lambda x : A.\ \lambda y : B.\ e)\ v)\ e_1))}$$

Figure 6.2. Operational semantics of λI_{Σ}

6.2.2 Dynamic Semantics

The reduction relation $\hookrightarrow$ is shown in Figure 6.2. Similarly to $\lambda I_{\leqslant}$, $\hookrightarrow$ is used for both type conversion in casts and term evaluation. The difference is that $\hookrightarrow$ in λI_{Σ} is *partially call-by-value*, which is mostly call-by-value along with several call-by-name rules. The reduction rules related to function applications are *rightmost call-by-value*, also called right-to-left call-by-value [Leroy 1990]. RP-Beta requires the argument to be a value. Arguments are reduced first in RP-AppR and functions can be further reduced in RP-AppL when arguments are values. Note that we use the metavariable v to range over values. The syntax of values is defined in Figure 6.1. As in call-by-value PITS, variables are values, which enables reduction with open terms (see Section 4.2.2).

Other reduction rules related to casts and dependent sums are *call-by-name*, which do not reduce sub-terms to values as in $\lambda I_{\leqslant}$. RP-Proj1 only accepts a dependent sum that is a value where its first component is a value. RP-Fst reduces sub-terms of first projections but there is no such rule for second projections. This is due to a *value restriction* [Swamy et al. 2011; Sjöberg et al. 2012]: we only treat $\langle v, e\ \mathsf{as}\ \Sigma x : A.\ B\rangle$ and $v.2$ as well-typed terms. RP-Proj2 and RP-Open both add two consecutive $\mathsf{cast}_{\uparrow}$ operators (also written as $\mathsf{cast}^2_{\uparrow}$) in reduced terms to preserve types. We will discuss value restriction and typing related issues later in Section 6.2.3.

The major reason of using the non-standard partially call-by-value semantics is to ensure type-safety, especially the subtype preservation lemma (see Lemma 6.3.9). Using the rightmost

style of call-by-value semantics is not a necessity but simplifies the proof for subtype preservation. The common leftmost/left-to-right call-by-value semantics can also be used in λI_{Σ} but at the expense of more complex proofs and incompatibility with other language features such as bounded quantification. Moreover, there are other useful applications of the rightmost variant, e.g., for an efficient implementation of multiple application in ML [Leroy 1990]. We will discuss these issues later in Section 6.3.3.

6.2.3 Static Semantics

Figure 6.3 shows the static semantics of λI_{Σ}, i.e., unified subtyping $\Gamma \vdash e_1 \leqslant e_2 : A$ and context well-formedness $\vdash \Gamma$. Similarly to $\lambda I_{\leqslant}$, unified subtyping is a single judgment that combines both typing and subtyping. The typing judgment $\Gamma \vdash e : A$ is syntactic sugar for $\Gamma \vdash e \leqslant e : A$. The context well-formedness judgment $\vdash \Gamma$ ensures every newly added type binding $x : A$ is fresh and A is a well-formed type. Note that there is no bounded quantification in λI_{Σ} and only type binding is supported in Γ. For the unified subtyping judgment, we focus on the discussion of new rules about dependent sums and other differences to $\lambda I_{\leqslant}$, e.g., value restriction.

Variables and Top Types. As bounded quantification is not supported in λI_{Σ}, assumptions about subtyping cannot be added into the context. Thus, there is no subtyping of variables as indicated by SP-Var, i.e., the subtype of a variable is itself. As mentioned in Section 6.2.1, top types in λI_{Σ} are now just ordinary types with kind $\star$ as in SP-Top and SP-TopRefl. The simplification of rules for variables and top types also eases the metatheory proof, as discussed in Section 6.3.1.

Value Restriction. Unlike the call-by-name semantics of $\lambda I_{\leqslant}$, we use a (partially) call-by-value semantics in λI_{Σ} and employ *value restriction* as in call-by-value PITS to ensure type preservation in a simple way (see Section 4.2.1). There are two rules for function applications, SP-App for non-dependent ones and SP-AppV for dependent ones. Dependent functions can only be applied to value arguments, while there is no restriction for non-dependent functions. Also, variables are treated as values (see Figure 6.1) as in call-by-value PITS, which enables reduction of *open terms* (e.g. $(\lambda x : Int.\ x)\ y$) in call-by-value semantics. As a result, when doing substitutions, substitutes must be values in order to maintain the computational behavior of variables.

Dependent Sums. SP-Sigma checks the well-formedness of dependent sum types (i.e. Sigma-types). The binder x can show in the body B_1 or B_2. Like rule SP-Abs for subtyping functions, we limit the subtyping of Sigma-types to be *pointwise*, which requires the types of binders to be identical. This is different from subtyping Pi-types in rule SP-Prod, where the argument types are contravariant.

SP-Pair defines the constructor of dependent sums, which also uses pointwise subtyping that fixes the first component. A dependent sum allows the first component (v) to show in the type of the second component (e_1 and e_2). The type of the second component is a substitution $B[x \mapsto v]$ and the substitute v, i.e., the first component, must be a value since variables are values (see Figure 6.1).

Projections. SP-Fst and SP-Snd are the rules for first and second projection, respectively. Allowing the typing of second projection indicates dependent sums in λI_{Σ} are *strong*. In rule SP-Snd, we follow the design of pointwise subtyping and disable the subtyping of second projection

$\boxed{\Gamma \vdash e_1 \leqslant e_2 : A}$ *(Unified Subtyping)*

SP-Ax
$$\frac{\vdash \Gamma}{\Gamma \vdash \star \leqslant \star : \star}$$

SP-Var
$$\frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x \leqslant x : A}$$

SP-Top
$$\frac{\Gamma \vdash e : \star}{\Gamma \vdash e \leqslant \top : \star}$$

SP-TopRefl
$$\frac{\vdash \Gamma}{\Gamma \vdash \top \leqslant \top : \star}$$

SP-Abs
$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash e_1 \leqslant e_2 : B \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash (\lambda x : A.\ e_1) \leqslant (\lambda x : A.\ e_2) : \Pi x : A.\ B}$$

SP-App
$$\frac{\Gamma \vdash e_1 \leqslant e_2 : A \to B \qquad \Gamma \vdash e_3 : A \qquad \Gamma \vdash B : \star}{\Gamma \vdash e_1\ e_3 \leqslant e_2\ e_3 : B}$$

SP-AppV
$$\frac{\Gamma \vdash e_1 \leqslant e_2 : \Pi x : A.\ B \qquad \Gamma \vdash v : A}{\Gamma \vdash e_1\ v \leqslant e_2\ v : B[x \mapsto v]}$$

SP-Prod
$$\frac{\Gamma \vdash A_2 \leqslant A_1 : \star \qquad \Gamma, x : A_1 \vdash B_1 : \star \qquad \Gamma, x : A_2 \vdash B_1 \leqslant B_2 : \star}{\Gamma \vdash (\Pi x : A_1.\ B_1) \leqslant (\Pi x : A_2.\ B_2) : \star}$$

SP-Castup
$$\frac{\Gamma \vdash B : \star \qquad \Gamma \vdash e_1 \leqslant e_2 : A \qquad B \hookrightarrow A}{\Gamma \vdash \mathsf{cast}_{\uparrow}\ e_1 \leqslant \mathsf{cast}_{\uparrow}\ e_2 : B}$$

SP-Castdn
$$\frac{\Gamma \vdash e_1 \leqslant e_2 : A \qquad A \hookrightarrow B}{\Gamma \vdash \mathsf{cast}_{\downarrow}\ e_1 \leqslant \mathsf{cast}_{\downarrow}\ e_2 : B}$$

SP-Sigma
$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B_1 \leqslant B_2 : \star}{\Gamma \vdash (\Sigma x : A.\ B_1) \leqslant (\Sigma x : A.\ B_2) : \star}$$

SP-Pair
$$\frac{\Gamma \vdash v : A \qquad \Gamma \vdash e_1 \leqslant e_2 : B[x \mapsto v] \qquad \Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \langle v, e_1\ \mathsf{as}\ \Sigma x : A.\ B\rangle \leqslant \langle v, e_2\ \mathsf{as}\ \Sigma x : A.\ B\rangle : \Sigma x : A.\ B}$$

SP-Fst
$$\frac{\Gamma \vdash e_1 \leqslant e_2 : \Sigma x : A.\ B}{\Gamma \vdash e_1.1 \leqslant e_2.1 : A}$$

SP-Snd
$$\frac{\Gamma \vdash v : \Sigma x : A.\ B}{\Gamma \vdash v.2 \leqslant v.2 : (\lambda x : A.\ B)\,(v.1)}$$

SP-Open
$$\frac{\begin{array}{c}\Gamma \vdash v : \Sigma x : A.\ B \\ \Gamma, x : A, y : B \vdash e : C \qquad \Gamma \vdash A : \star \\ \Gamma, x : A \vdash B : \star \qquad \Gamma, x : A \vdash C : \star\end{array}}{\Gamma \vdash \mathbf{open}\ v\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ e \leqslant \mathbf{open}\ v\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ e : (\lambda x : A.\ C)\,(v.1)}$$

SP-Sub
$$\frac{\Gamma \vdash e_1 \leqslant e_2 : A \qquad \Gamma \vdash A \leqslant B : \star}{\Gamma \vdash e_1 \leqslant e_2 : B}$$

$\boxed{\vdash \Gamma}$ *(Well-formedness)*

WP-Nil
$$\frac{}{\vdash \varnothing}$$

WP-Cons
$$\frac{\Gamma \vdash A : \star \qquad x \text{ fresh in } \Gamma}{\vdash \Gamma, x : A}$$

Syntactic Sugar $\qquad \Gamma \vdash e : A \triangleq \Gamma \vdash e \leqslant e : A$

Figure 6.3. Static semantics of λI_{Σ}

by requiring both sides of the relation to be the same. We also employ value restriction and require the dependent sum to be a value. The reason is similar to rule SP-AppV. The typing result of SP-Snd is dependent on the form of input terms. Pointwise subtyping simplifies the rule definition (see Section 5.2.3). Restricting input terms as values forbids the reduction of sub-terms in second projection, which ensures type preservation (see Section 4.2.1).

Assume that we drop the value restriction for typing second projections and allow reducing

the sub-terms of second projections as follows:

$$\frac{e \hookrightarrow e'}{e.2 \hookrightarrow e'.2}$$

Then $e.2$ and $e'.2$ have different types due to lack of beta-equivalence, since their types contain e and e', respectively, which breaks type preservation.

Iso-strong sums use a different typing rule for the second projection. The typing of second projection is different from the standard definition:

$$\frac{\Gamma \vdash v : \Sigma x : A.\ B}{\Gamma \vdash v.2 : B[x \mapsto v.1]}$$

where the typing result is a substitution $B[x \mapsto v.1]$. As mentioned in Section 6.1.2, in order to avoid the use of full casts with parallel reduction, we do not use the standard typing rule above that involves direct type substitution. In SP-Snd, the typing result is a type-level *application*, i.e., $(\lambda x : A.\ B)\,(v.1)$. Note that the typing result is well-formed since $\lambda x : A.\ B$ is a non-dependent function where B has type $\star$ with no x. The type-level application can be viewed as an *intermediate step* before substitution. We can use explicit type casts to obtain the final substituted type. For example, consider v is a dependent sum $v = \langle v', e' \text{ as } \Sigma x : A.\ B\rangle$. Notice that the following reduction steps hold:

$$\begin{array}{rcl} \langle v', e' \text{ as } \Sigma x : A.\ B\rangle.1 & \hookrightarrow & v' \\ (\lambda x : A.\ B)\, v' & \hookrightarrow & B[x \mapsto v'] \end{array}$$

We can add two consecutive $\mathsf{cast}_{\downarrow}$ operators to obtain the substituted type:

$$\begin{array}{rcl} v.2 & : & (\lambda x : A.\ B)\,(\langle v', e' \text{ as } \Sigma x : A.\ B\rangle.1) \\ \mathsf{cast}^2_{\downarrow}\, v.2 & : & B[x \mapsto v'] \end{array}$$

The type of $\mathsf{cast}^2_{\downarrow}\, v.2$ follows the premise in SP-Pair for typing second component. Recall that the reduction rule for second projection, i.e., RP-Proj2, adds two $\mathsf{cast}_{\uparrow}$s, which exactly balance the manually added two $\mathsf{cast}_{\downarrow}$s:

$$\begin{array}{rcl} v.2 & \hookrightarrow & \mathsf{cast}^2_{\uparrow}\, v' \\ \mathsf{cast}^2_{\downarrow}\, v.2 & \hookrightarrow & \mathsf{cast}^2_{\downarrow}\,(\mathsf{cast}^2_{\uparrow}\, v') \hookrightarrow v' \end{array}$$

Thus, we can achieve *type-preserving* reduction for the second projection through explicit casts.

Openings. SP-Open is the rule for the opening operation, an alternative to projection for destructing a dependent sum. It has the same restrictions as SP-Snd: value restriction and no subtyping. The typing result is also similar, which adopts the approach of iso-strong sums and is a type-level application. Two pattern variables x and y correspond to the first and second component of a dependent sum v, respectively. The inner term e is typed as C against pattern variables with typing $x : A$ and $y : B$. Note that we only allow x to show in C but not y, which is checked by the last premise. Such restriction ensures the typing result, i.e., $(\lambda x : A.\ C)\,(v.1)$, will not be out of scope. Despite of such restriction, the opening operation is still more expressive

than the ordinary existential opening that even does not allow x to show in C. Opening operation can be seen as a generalization of second projection:

$$v.2 = \mathbf{open}\ v\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ y$$

where both sides have the same typing result. Opening allows *directly* using the type of the second component as a pattern variable y inside e, while the type of second projection is an application that cannot be directly used. For example, imagine that v is a dependent sum whose second component is a function that takes an integer:

$$v : \Sigma x : \star.\ Int \to x$$

The second projection of v has type

$$v.2 : (\lambda x : \star.\ Int \to x)\,(v.1)$$

which cannot take an integer, say $(v.2)\,3$. In λI_{Σ}, we use opening instead:

$$\mathbf{open}\ v\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ (y\,3) : (\lambda x : \star.\ x)\,(v.1)$$

where the pattern y can take an integer since $y : Int \to x$.

No Nested Second Projections. The value restriction simplifies the metatheory but unfortunately limits the ability of nesting *second* projections in λI_{Σ}. For example, $(v.2).2$ is not a well-formed term since $v.2$ is not a value. Furthermore, we cannot use openings to simulate nested second projections either. Consider the following example:

$$\begin{array}{lcl} v & : & \Sigma x_1 : A_1.\ (\Sigma x_2 : A_2.\ B) \\ \mathbf{open}\ v\ \mathbf{as}\ \langle x_1, \boldsymbol{y}\rangle\ \mathbf{in} & : & (\,\lambda x_1 : A_1. \\ \quad \mathbf{open}\ \boldsymbol{y}\ \mathbf{as}\ \langle x_2, z\rangle\ \mathbf{in}\ z & & \quad (\lambda x_2 : A_2.\ B)\ (\boldsymbol{y}.1)\ )\ (v.1) \end{array}$$

We simulate nested second projections $(v.2).2$ by two consecutive openings. However, the typing result is not well-formed since the pattern variable y leaks out of the scope. This violates the restriction imposed by the last premise of SP-Open, which requires that the second pattern variables (e.g. y) should not show in the result type.

Notice that nested *first* projections are still allowed. Iso-strong sums are still more expressive than weak sums and sufficient for our purposes to encode interesting examples such as traits with type members (see Section 6.1.3). Restricting nested projections is also used by other calculi to simplify the presentation and metatheory. For example, DOT [Rompf and Amin 2016; Amin et al. 2016] only supports a single level of type selection on variables such as $x.A$. Removing value restriction and supporting nested second projections are left as future work (see Section 8.2).

No Direct Subsumption of Non-dependent Pairs. Finally, due to the non-standard typing in SP-Snd and SP-Open, iso-strong sums in λI_{Σ} cannot *directly* subsume non-dependent pairs or ordinary weak existentials. Consider a non-dependent pair $v = \langle 1, 2\ \mathsf{as}\ \Sigma x : Int.\ Int\rangle$. By the standard typing rule, the type of $v.2$ is $Int[x \mapsto v.1] = Int$ where the substitution has no effect and the typing result is the same as non-dependent pairs. In contrast, by SP-Snd we have

$v.2 : (\lambda x : Int.\ Int)\, v.1$, where the typing result is still a type-level application but not Int. Nonetheless, we can add explicit casts to recover the standard typing results as examples above (see also Sections 6.1.2 and 6.1.3). Alternatively, we can just encode typing rules for non-dependent pairs and weak existentials using standard Church encodings [Pierce 2002].

6.3 Metatheory of λI_Σ

We discuss the metatheory of the λI_Σ calculus, mainly on two results: transitivity and type-safety. The proof process is mostly similar to the one of $\lambda I_\leqslant$ (see Section 5.3). However, due to differences of dynamic and static semantic, especially call-by-value reduction and dependent sums, λI_Σ requires additional lemmas and some changes in proof strategies. We will focus on discussion of such changes from the metatheory of $\lambda I_\leqslant$. Note that we have not proved the decidability of λI_Σ yet, though we believe the property should hold and leave it as future work (see Section 8.2).

6.3.1 Basic Lemmas

We first discuss the changes to basic lemmas compared to $\lambda I_\leqslant$. Due to the absence of bounded quantification, some basic lemmas now can be proved independently, such as reflexivity and type narrowing.

Reflexivity. The reflexivity lemma of unified subtyping is defined as follows:

Lemma 6.3.1 (Reflexivity). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$, *then both* $\Gamma \vdash e_1 : A$ *and* $\Gamma \vdash e_2 : A$ *hold.*

In $\lambda I_\leqslant$, reflexivity is proved by splitting into two sub-lemmas, left and right reflexivity (Lemma 5.3.2 and 5.3.3). Such proof strategy is necessary because right reflexivity has a mutual dependency on another lemma, correctness of types (Lemma 5.3.11). The mutual dependency is caused by generalized top type (see discussion in Section 5.3.1). In λI_Σ, such mutual dependency does not exist because top type is an ordinary type but not generalized:

$$\frac{\Gamma \vdash e : \star}{\Gamma \vdash e \leqslant \top : \star}\ \textsc{SP-Top} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \top \leqslant \top : \star}\ \textsc{SP-TopRefl}$$

For cases SP-Top and SP-TopRefl, reflexivity can be shown immediately from the induction hypothesis. Thus, we can prove the reflexivity lemma first without relying on other results.

Weakening, Narrowing and Substitution. The weakening lemma can be proved similarly as in $\lambda I_\leqslant$ by induction on the derivation of $\Gamma_1, \Gamma_3 \vdash e_1 \leqslant e_2 : A$:

Lemma 6.3.2 (Weakening). *If* $\Gamma_1, \Gamma_3 \vdash e_1 \leqslant e_2 : A$ *and* $\vdash \Gamma_1, \Gamma_2, \Gamma_3$, *then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash e_1 \leqslant e_2 : A$.

Due to the absence of bounded quantification, there is no subtyping binding in the context. The narrowing lemma in λI_Σ has just one form, type narrowing:

Lemma 6.3.3 (Type Narrowing). *Given* $\Gamma_1, x : B, \Gamma_2 \vdash e_1 \leqslant e_2 : C$, *if* $\Gamma_1 \vdash A \leqslant B : \star$ *and* $\Gamma_1 \vdash e : A$, *then* $\Gamma_1, x : A, \Gamma_2 \vdash e_1 \leqslant e_2 : C$.

The proof is by induction on the derivation of $\Gamma_1, x : B, \Gamma_2 \vdash e_1 \leqslant e_2 : C$. Also notice that the subtyping of variables in λI_Σ is only a reflexive relation (see Section 6.2.3). There is no transitivity rule for variables, such as S-VarTrans in $\lambda I_\leqslant$. The proof does not rely on the transitivity lemma or consistency of typing lemma (Lemma 5.3.5), a special case of transitivity. Finally, the substitution lemma in λI_Σ has a different form:

Lemma 6.3.4 (Substitution). *If* $\Gamma_1, x : B, \Gamma_2 \vdash e_1 \leqslant e_2 : A$ *and* $\Gamma_1 \vdash v : B$, *then* $\Gamma_1, \Gamma_2[x \mapsto v] \vdash e_1[x \mapsto v] \leqslant e_2[x \mapsto v] : A[x \mapsto v]$.

where the substitute should be a value v, since we treat variables as values and allow reduction of open terms, as mentioned in Section 6.2.3. Such form is similar to the one in call-by-value PITS (see Lemma 4.2.1). The proof relies on transitivity as in $\lambda I_\leqslant$, as well as two additional lemmas, *value substitution* and *reduction substitution* due to value restriction in rules:

Lemma 6.3.5 (Value Substitution)**.**

If v_1 *and* v_2 *are values, then* $v_1[x \mapsto v_2]$ *is still a value.*

Lemma 6.3.6 (Reduction Substitution)**.**

If v *is a value and* $e_1 \hookrightarrow e_2$ *holds, then* $e_1[x \mapsto v] \hookrightarrow e_2[x \mapsto v]$ *still holds.*

Proofs of two lemmas are straightforward by induction on the structure of v_1 and derivation of $e_1 \hookrightarrow e_2$, respectively. Note that although the proof of substitution lemma still depends on transitivity, this is only one-way dependency. Thus, we can similarly show substitution later after proving transitivity as in $\lambda I_\leqslant$.

6.3.2 Transitivity

The unified subtyping rules in λI_Σ do not contain a built-in transitivity rule. We need to prove the transitivity property. Due to the presence of the subsumption rule (RP-Sub), we need to prove a *generalized* transitivity property as in $\lambda I_\leqslant$ (see Section 5.3.2). However, we use a slightly different generalization:

Lemma 6.3.7 (Generalized Transitivity). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$ *and* $\Gamma \vdash e_2 \leqslant e_3 : B$, *then* $\Gamma \vdash e_1 \leqslant e_3 : B$.

which is a *rightmost* generalization. The difference is the type of the conclusion judgment: we use $\Gamma \vdash e_1 \leqslant e_3 : B$ in λI_Σ, while $\Gamma \vdash e_1 \leqslant e_3 : A$ in $\lambda I_\leqslant$ which is a *leftmost* generalization. The reason is that the typing of the top type ($\top$) is different. In $\lambda I_\leqslant$, the top type is generalized to be any type by rule S-Top, while it is an ordinary type of kind $\star$ in λI_Σ by rule SP-Top:

$$\text{S-Top}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash e \leqslant \top : A} \qquad \text{SP-Top}\ \frac{\Gamma \vdash e : \star}{\Gamma \vdash e \leqslant \top : \star}$$

Consider the case that $e_3 = \top$ and we have $\Gamma \vdash e_1 \leqslant e_2 : A$ and $\Gamma \vdash e_2 \leqslant \top : B$. For the generalized top type, we know $\Gamma \vdash e_1 : A$ by reflexivity from the former judgment. By S-Top, we obtain the target judgment $\Gamma \vdash e_1 \leqslant \top : A$, which has the same type as the leftmost term e_1. For the ordinary top type, we know $B = \star$ by inversion of SP-Top. Note that the target is the subtyping relation between e_1 and $\top$, which requires the type to be $\star$ by SP-Top. It is natural

to set the target as $\Gamma \vdash e_1 \leqslant \top : B$ which uses the rightmost type of $\top$. Otherwise, if we set the target type to be the leftmost one, i.e., $\Gamma \vdash e_1 \leqslant \top : A$, we ought to show $A = \star$ which is unknown to be true. Thus, using rightmost generalized transitivity in λI_Σ simplifies the proof for the case of ordinary top type.

We prove the rightmost generalized transitivity by the same approach used in the leftmost one (see Section 5.3.2). The proof is by induction on the size of e_2 and an inner induction on the derivation of the first judgment $\Gamma \vdash e_1 \leqslant e_2 : A$. The case for Pi-types (SP-Prod) also needs the type narrowing lemma (Lemma 6.3.3). Then we can immediately show the original transitivity lemma which is a direct corollary:

Lemma 6.3.8 (Transitivity). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$ *and* $\Gamma \vdash e_2 \leqslant e_3 : A$, *then* $\Gamma \vdash e_1 \leqslant e_3 : A$.

6.3.3 Type Safety

Type-safety of λI_Σ is shown by the standard type preservation and progress lemmas [Wright and Felleisen 1994]. We discuss the design choices of one-step reduction for keeping subtype preservation and focus on the change of proofs due to the call-by-value semantics used in λI_Σ.

Partially Call-by-value. As in $\lambda I_\leqslant$, we need to show a generalized preservation lemma for the unified subtyping judgment, i.e., *subtype preservation*:

Lemma 6.3.9 (Subtype Preservation). *If* $\Gamma \vdash e_1 \leqslant e_2 : A$, $e_1 \hookrightarrow e_1'$, $e_2 \hookrightarrow e_2'$, *then* $\Gamma \vdash e_1' \leqslant e_2' : A$.

To ensure that such property holds, we need to restrict the call-by-value semantics to be *partial* (see Section 6.2.2). We only allow call-by-value semantics for function applications, but call-by-name rules for other constructs, e.g., no reduction of sub-terms in casts or dependent sums. Otherwise, call-by-value rules for casts and dependent sums would break subtype preservation. We show two counter-examples.

Counter-examples for Full Call-by-value. Assume that we treat $\mathsf{cast}_\uparrow\ v$ as a value and allow reducing the sub-term in $\mathsf{cast}_\uparrow$ by the following rule:

$$\frac{e \hookrightarrow e'}{\mathsf{cast}_\uparrow\ e \hookrightarrow \mathsf{cast}_\uparrow\ e'}$$

Let $e_1 = \mathsf{cast}_\downarrow\,(\mathsf{cast}_\uparrow\,(\mathsf{cast}_\uparrow\ e))$ and $e_2 = \mathsf{cast}_\downarrow\,(\mathsf{cast}_\uparrow\,(\mathsf{cast}_\uparrow\ \top))$ and assume there exists e' such that $e \hookrightarrow e'$. Note that $\mathsf{cast}_\uparrow\ e$ is reducible and $\mathsf{cast}_\uparrow\ \top$ is a value. Relations in the subtype preservation lemma are as follows:

$$\begin{array}{ccc} \mathsf{cast}_\downarrow\,(\mathsf{cast}_\uparrow\,(\mathsf{cast}_\uparrow\ e)) & \leqslant & \mathsf{cast}_\downarrow\,(\mathsf{cast}_\uparrow\,(\mathsf{cast}_\uparrow\ \top)) \\ \downarrow & & \downarrow \\ \mathsf{cast}_\downarrow\,(\mathsf{cast}_\uparrow\,(\mathsf{cast}_\uparrow\ e')) & \not\leqslant & \mathsf{cast}_\uparrow\ \top \end{array}$$

The target subtyping relation would not hold because the outer most constructs of both sides are different, i.e., left side is $\mathsf{cast}_\downarrow$ left, while right side is $\mathsf{cast}_\uparrow$.

Assume that we treat $\langle v_1, v_2 \text{ as } \Sigma x : A.\ B\rangle$ as a value and allow reducing the second component of a dependent sum by the following rule:

$$\frac{e \hookrightarrow e'}{\langle v, e \text{ as } \Sigma x : A.\ B\rangle \hookrightarrow \langle v, e' \text{ as } \Sigma x : A.\ B\rangle}$$

Let $e_1 = \langle \star, e \text{ as } \Sigma x : \star.\ \star\rangle.1$ and $e_2 = \langle \star, \top \text{ as } \Sigma x : \star.\ \star\rangle.1$ and assume there exists e' such that $e \hookrightarrow e'$. Note that $\langle \star, e \text{ as } \Sigma x : \star.\ \star\rangle$ is reducible, while $\langle \star, \top \text{ as } \Sigma x : \star.\ \star\rangle$ is a value. The following diagram show the relations in the subtype preservation lemma:

$$\begin{array}{ccc} \langle \star, e \text{ as } \Sigma x : \star.\ \star\rangle.1 & \leqslant & \langle \star, \top \text{ as } \Sigma x : \star.\ \star\rangle.1 \\ \downarrow & & \downarrow \\ \langle \star, e' \text{ as } \Sigma x : \star.\ \star\rangle.1 & \not\leqslant & \star \end{array}$$

Apparently, the target subtyping relation does not hold, since the only subtype of $\star$ is itself.

The Problem of Leftmost Call-by-value. Another unusual design of one-step reduction ($\hookrightarrow$) is the adoption of *rightmost* call-by-value semantics (see Section 6.2.2). In function applications, we first reduce all arguments to values and then reduce the functions, as indicated by the following reduction rules:

$$\text{RP-AppR}\ \frac{e_2 \hookrightarrow e_2'}{e_1\ e_2 \hookrightarrow e_1\ e_2'} \qquad \text{RP-AppL}\ \frac{e \hookrightarrow e'}{e\ v \hookrightarrow e'\ v}$$

Alternatively, one can use the following the *leftmost* variant of reduction rules that reduce the functions first:

$$\frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \frac{e_2 \hookrightarrow e_2'}{v\ e_2 \hookrightarrow v\ e_2'}$$

The two variants have different impact on the proof of subtype preservation lemma, especially when disproving two *diverging cases*. Unlike the weak-head call-by-name reduction in $\lambda I_{\leqslant}$ that only reduces the function part, it is possible in λI_{Σ} that the reduction happens on different positions within an application and diverges on two sides of a subtyping relation. We need to rule out such cases, which would break the subtype preservation.

For the leftmost variant, the diverging cases can be illustrated by the following diagram:

$$\begin{array}{ccc} e_1\ e_2 & \leqslant & v\ e_2 \\ \downarrow & & \downarrow \\ e_1'\ e_2 & \not\leqslant & v\ e_2' \end{array} \qquad \begin{array}{ccc} v\ e_2 & \leqslant & e_1\ e_2 \\ \downarrow & & \downarrow \\ v\ e_2' & \not\leqslant & e_1'\ e_2 \end{array}$$

(1) (2)

where we assume $e_1 \hookrightarrow e_1'$ and $e_2 \hookrightarrow e_2'$. Note that $e_2' \neq e_2$ and subtyping of applications is pointwise in rules SP-App and SP-AppV. Both targets in (1) and (2) do not hold.

In case (1), the function part e_1 on the left-hand side is still reducible, while the right-hand side term $v\ e_2$ only has the reducible argument e_2. Thus, the reduction happens in different places: the function position on the left-hand side and the argument position on the right-hand side. Case (2) is simply the mirror of case (1), which exchanges the sides of the subtyping relation.

By inversion of the condition, we know $e_1 \leqslant v$ and $v \leqslant e_1$ from case (1) and (2), respectively. We can disprove case (1) and (2) by show that e_1 is a value which is not reducible:

Lemma 6.3.10 (Value Preservation)**.**

1. *If* $\Gamma \vdash e_1 \leqslant e_2 : (\Pi x : A.\ B)$ *and* e_2 *is a value, then* e_1 *is also a value.*
2. *If* $\Gamma \vdash e_1 \leqslant e_2 : A$ *and* e_1 *is a value, then* e_2 *is also a value.*

Thus, both diverging cases are impossible and will not break the subtype preservation lemma. Notice that the proof depends on several specific rules of λI_Σ. The first sub-lemma relies on the fact that T has kind $\star$ by SP-Top. The second sub-lemma relies on the subtyping rule SP-Var which limits subtyping of variables to be reflexive only. For the first sub-lemma, if we use the generalized top type that can have a Pi-type, when $e_2 = \top$, e_1 can be any well-typed function that may not be a value. For the second sub-lemma, if bounded quantification is allowed, when $e_1 = x$, e_2 is not necessarily a value but can be any term as long as $x \leqslant e_2 : A \in \Gamma$. If we want to extend λI_Σ with generalized top type or bounded quantification like $\lambda I_{\leqslant}$, the value preservation lemma will not hold and we may not be able to disprove the diverging cases of subtype preservation for the leftmost variant.

Rightmost Call-by-value to the Rescue. For the rightmost variant, the diverging cases are simply not possible. If we try to follow the diagram of the leftmost variant, assuming that $e_1 \hookrightarrow e_1'$, $e_2 \hookrightarrow e_2'$, we have

$$\begin{array}{ccc} e_1\ v & \leqslant & e_2\ v \\ \downarrow & & \not\downarrow \\ e_1'\ v & \not\leqslant & e_2\ e \end{array} \qquad (3) \qquad\qquad \begin{array}{ccc} e_1\ v & \leqslant & e_2\ v \\ \not\downarrow & & \downarrow \\ e_1\ e & \not\leqslant & e_2'\ v \end{array} \qquad (4)$$

In (3) and (4), the arguments are already values due to the rightmost reduction strategy. They also should be the same due to the pointwise subtyping rules. Then, the reduction of the argument part, i.e., $v \hookrightarrow e$, is contradictory, since values are not reducible.

Thus, the rightmost call-by-value strategy is more suitable for meta-theoretical development in λI_Σ. It naïvely rules out the diverging cases and simplifies the proof of subtype preservation. It is also less restrictive and does not rely on the value preservation lemma, which makes it possible to extend λI_Σ with features such as bounded quantification and generalized top type without breaking subtype preservation.

Determinacy of Reduction. Though the one-step reduction is changed to use the (partially) call-by-value semantics, it still has the same determinacy property as the call-by-name variant in $\lambda I_{\leqslant}$:

Lemma 6.3.11 (Determinacy of Reduction)**.** *If* $e \hookrightarrow e_1$ *and* $e \hookrightarrow e_2$*, then* $e_1 = e_2$*.*

The proof holds few surprises, which is done by induction on the derivation of $e \hookrightarrow e_1$ and inversion on $e \hookrightarrow e_2$.

Preservation and Progress. Similarly to $\lambda I_{\leqslant}$, we prove a generalized subtype preservation lemma to avoid induction hypothesis issues in the $\mathsf{cast}_\uparrow$ case:

Lemma 6.3.12 (Generalized Subtype Preservation)**.** *Given that* $\Gamma \vdash e_1 \leqslant e_2 : A$ *holds,*

1. *if* $A \hookrightarrow A'$ *and both* e_1 *and* e_2 *are* $\mathsf{cast}_\uparrow$ *terms, i.e.,* $e_1 = \mathsf{cast}_\uparrow\, e_1'$ *and* $e_2 = \mathsf{cast}_\uparrow\, e_2'$*, then* $\Gamma \vdash e_1' \leqslant e_2' : A'$*;*

2. *otherwise, if* $e_1 \hookrightarrow e_1'$ *and* $e_2 \hookrightarrow e_2'$*, then* $\Gamma \vdash e_1' \leqslant e_2' : A$*.*

The first case is slightly changed since we drop the annotation of $\mathsf{cast}_\uparrow$. The proof is by induction on the derivation of $\Gamma \vdash e_1 \leqslant e_2 : A$, which is similar to the one of $\lambda I_{\leqslant}$. The diverging cases can be trivially disproved due to the choice of rightmost call-by-value reduction. The proof similarly depends on the "reduction in the middle" lemma, which holds for one-step call-by-value reduction:

Lemma 6.3.13 (Reduction Exists in the Middle)**.** *Given that* $\Gamma \vdash C \leqslant B : D$ *and* $\Gamma \vdash B \leqslant A : D$*, if* $C \hookrightarrow C'$ *and* $A \hookrightarrow A'$*, then there exists* B' *such that* $B \hookrightarrow B'$*.*

Finally, we can conclude the type preservation lemma, which is simply a corollary of subtype preservation:

Lemma 6.3.14 (Type Preservation)**.** *If* $\Gamma \vdash e : A$ *and* $e \hookrightarrow e'$*, then* $\Gamma \vdash e' : A$*.*

We prove the progress lemma that has the judgment with an empty context:

Lemma 6.3.15 (Progress)**.** *If* $\varnothing \vdash e : A$ *then either* e *is a value* v *or there exists* e' *such that* $e \hookrightarrow e'$*.*

Unlike $\lambda I_{\leqslant}$, we do not prove a generalized progress lemma (Lemma 5.3.18) since the inert terms (see Section 5.2.2), which cover all stuck open terms, are not defined for the call-by-value semantics in λI_{Σ}.

6.4 The Sig Language

We formally present the surface language namely **Sig**, which supports Scala-like first-class traits with *type members*. **Sig** is a lightweight layer built on top of λI_{Σ} for presenting how iso-strong sums in λI_{Σ} can encode complex constructs such as traits. Expressions of **Sig** will be elaborated into λI_{Σ} terms through a type-directed translation. **Sig** does not expose explicit cast operators. The translation process will automatically insert necessary casts to keep terms well-typed. In spite of many language limitations in **Sig**, the main purpose of **Sig** is to show the application of iso-strong sums and the inference of casts via the elaboration semantics. In the rest of this section, we will show the syntax, static semantics, type-directed translation and translation soundness of **Sig**.

6.4.1 Syntax

Figure 6.4 shows the syntax of **Sig**, which is unified as λI_{Σ}. Terms (E) and types (T) are in the same syntactic category. Basic language constructs can be directly mapped to λI_{Σ} terms, e.g., Type corresponds to $\star$ and the function type corresponds to a Pi-type, etc. We use the syntactic sugar $T_1 \to T_2$ for non-dependent function types. Note that there are no explicit type cast operators in **Sig**.

Expressions	E, T	$::=$	$x \mid \mathsf{Type} \mid (x : T_1) \to T_2 \mid E_1\ E_2 \mid \lambda(x : T) \Rightarrow E$
		$\mid$	$\mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} \mid \mathbf{obj}\,\{\,\mathsf{type}\,L = E; M\}\,\mathbf{as}\,T$
		$\mid$	$x.l \mid x.L$
Trait Bindings	S	$::=$	$\mathsf{val}\,l_1 : T_1; \ldots; \mathsf{val}\,l_n : T_n$
Object Bindings	M	$::=$	$\mathsf{val}\,l_1 = E_1; \ldots; \mathsf{val}\,l_n = E_n$
Contexts	Δ	$::=$	$\varnothing \mid \Delta, x : T \mid \Delta, x = \langle x_1, x_2\rangle : T$
Values	V	$::=$	$\mathsf{Type} \mid x \mid (x : T_1) \to T_2 \mid \lambda(x : T) \Rightarrow E$
		$\mid$	$\mathbf{obj}\,\{\,\mathsf{type}\,L = E; M\}\,\mathbf{as}\,T \mid x.L$
Syntactic Sugar	$T_1 \to T_2$	$\triangleq$	$(x : T_1) \to T_2 \quad \text{where } x \notin \mathsf{FV}(T_2)$

Figure 6.4. Syntax of **Sig**

Traits and Objects. A trait has the syntax $\mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\}$ which is a compound type that has one type member L and multiple value members. The bindings of value members S have the form $\mathsf{val}\,l_i : T_i$, which means the label l_i has type T_i. Only the type member L but not value members l_i can show in T_i. We use the upper-case metavariable L for the label of type member, and the lower-case l for the label of value members. Objects are instances of traits. The syntax of objects is $\mathbf{obj}\,\{\,\mathsf{type}\,L = E; M\}\,\mathbf{as}\,T$ where E is the actual type member implementation for the abstract label L. M is the object bindings with the form $\mathsf{val}\,l_i = E_i$ where E_i is the implementation of the member l_i. Type T after the keyword **as** is the type annotation. The member access of objects is denoted by $x.L$ and $x.l$, which are for the type member and value member, respectively.

Restrictions on Traits. For simplicity reasons, we have several restrictions related to traits:

- There is only *one* type member in traits, so that traits can be directly mapped to Sigma-types of λI_Σ which have just one binder. Nonetheless, we can simulate multiple type members. For example, noting that the type T of the type member is polymorphic, which is not limited to Type, we can specify T as a compound type such as records to represent multiple type members.
- The member access operations are bound only to variables (e.g. $x.L$) but not paths (e.g. $x.y.L$). This is due to the value restriction in the target language λI_Σ. The restriction of member access is also commonly used in other calculi for a simpler formalization of traits, such as Dependent Object Types (DOT) [Rompf and Amin 2016; Amin et al. 2016].
- We do not support subtyping of type members in **Sig** because there is no bounded quantification in the target language.

6.4.2 Static Semantics

The static semantics of **Sig** is shown in Figure 6.5 and 6.6. The typing context Δ has two kinds of bindings, namely the type binding $x : T$ and trait binding $x = \langle x_1, x_2\rangle : T$ (see Figure 6.4). Typing rules only use the typing binding. The trait binding is used only for translation and will be discussed later in Section 6.4.3. The well-formedness of context is checked by the judgment $\vdash \Delta$. Object typing $\Delta \vdash M : S$ and trait well-formedness $\Delta \vdash S$ reuse the typing judgment for inferring the type of object bindings and checking if types of value members are well-formed, respectively. The typing judgment is denoted by $\Delta \vdash E : T$. Typing rules for basic constructs

$\boxed{\Delta \vdash E : T}$ *(Term Typing)*

$$\textsc{ST-Ax}\ \frac{\vdash \Delta}{\Delta \vdash \mathsf{Type} : \mathsf{Type}} \qquad \textsc{ST-Var}\ \frac{x : T \in \Delta \qquad \vdash \Delta}{\Delta \vdash x : T}$$

$$\textsc{ST-Pi}\ \frac{\Delta \vdash T_1 : \mathsf{Type} \qquad \Delta, x : T_1 \vdash T_2 : \mathsf{Type} \qquad T_2 \text{ is } x.L\text{-only-projection if } T_1 \text{ is a trait}}{\Delta \vdash (x : T_1) \to T_2 : \mathsf{Type}}$$

$$\textsc{ST-App}\ \frac{\Delta \vdash E_1 : T_1 \to T_2 \qquad \Delta \vdash E_2 : T_1}{\Delta \vdash E_1\ E_2 : T_2} \qquad \textsc{ST-AppV}\ \frac{\Delta \vdash E : (x : T_1) \to T_2 \qquad \Delta \vdash V : T_1 \qquad V \text{ is not an object}}{\Delta \vdash E\ V : T_2[x \mapsto V]}$$

$$\textsc{ST-AppM}\ \frac{\Delta \vdash E : (x : T_1) \to T_2 \qquad \Delta \vdash V_1 : T_1 \qquad T_1 = \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} \qquad V_1 = \mathbf{obj}\,\{\,\mathsf{type}\,L = V; M\}\ \mathbf{as}\ T_1}{\Delta \vdash E\ V_1 : T_2[x.L \mapsto V]}$$

$$\textsc{ST-Lam}\ \frac{\Delta \vdash T_1 : \mathsf{Type} \qquad \Delta, x : T_1 \vdash E : T_2 \qquad T_2 \text{ is } x.L\text{-only-projection if } T_1 \text{ is a trait}}{\Delta \vdash (\lambda(x : T_1) \Rightarrow E) : (x : T_1) \to T_2}$$

$$\textsc{ST-Mod}\ \frac{\Delta \vdash V : T \qquad \Delta \vdash M[L \mapsto V] : S[L \mapsto V] \qquad \Delta \vdash T_1 : \mathsf{Type} \qquad T_1 = \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} \qquad T \text{ is not a trait}}{\Delta \vdash \mathbf{obj}\,\{\,\mathsf{type}\,L = V; M\}\ \mathbf{as}\ T_1 : \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\}}$$

$$\textsc{ST-Sig}\ \frac{\Delta \vdash T : \mathsf{Type} \qquad \Delta, L : T \vdash S \qquad T \text{ is not a trait}}{\Delta \vdash \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} : \mathsf{Type}}$$

$$\textsc{ST-ProjT}\ \frac{\Delta \vdash x : \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\}}{\Delta \vdash x.L : T} \qquad \textsc{ST-Proj}\ \frac{\Delta \vdash x : \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} \qquad \mathsf{val}\,l : T_2 \in S}{\Delta \vdash x.l : T_2[L \mapsto x.L]}$$

Figure 6.5. Typing rules of **Sig**

are similar to ones in λI_{Σ}. There is no explicit or implicit type conversion rule. Also, there is *no subtyping relation* in **Sig** and thus no subsumption rule in the typing judgment. In the following text, we focus on typing rules related to traits.

Applications. Due to the value restriction of the target language, we have multiple application rules in **Sig**, namely ST-App, ST-AppV and ST-AppM. Similarly to λI_{Σ}, rules ST-App and ST-AppV are for non-dependent and dependent function application, respectively. The arguments to dependent functions need to be values V (see definitions in Figure 6.4). The case when arguments are objects is specifically handled by rule ST-AppM. We rule out such case in ST-AppV by requiring V not to be an object.

Rule ST-AppM checks if the argument V_1 is an object and T_1 in the function type is a trait type. The conclusion type is obtained by replacing all type member access $x.L$ in T_2 by the actual type member V. We use a special form of substitution $T_2[x.L \mapsto V]$, called *projection substitution*. It pattern-matches the member access form $x.L$ which contains two variables.

Additionally, to ensure the substitution result is well-formed, we need to restrict how x occurs in T_2, namely if x shows free in T_2, it must be in the form $x.L$. Then after projection substitution, we can guarantee that x will not be out of scope in the conclusion type. We formally define such restriction and call T_2 a *$x.L$-only-projection term*:

Definition 6.4.1 (Projection-Only Terms)**.** *We call E $x.L$-*only-projection *if x and L are free variables in E and $x \notin \mathsf{FV}(E[x.L \mapsto y])$ for any fresh variable $y \neq x$.*

$\boxed{\vdash \Delta}$ *(Context Well-formedness)*

$$\frac{}{\vdash \varnothing}\ \text{SW-Nil} \qquad \frac{\vdash \Delta \qquad \Delta \vdash T : \mathsf{Type} \qquad x \text{ fresh in } \Delta}{\vdash \Delta, x : T}\ \text{SW-Cons}$$

$\boxed{\Delta \vdash M : S}$ *(Object Typing)*

$$\frac{\overline{\Delta \vdash E_i : T_i}^{\,i}}{\Delta \vdash \overline{\mathsf{val}\ l_i = E_i}^{\,i} : \overline{\mathsf{val}\ l_i : T_i}^{\,i}}\ \text{SM-Bind}$$

$\boxed{\Delta \vdash S}$ *(Trait Well-formedness)*

$$\frac{\overline{\Delta \vdash T_i : \mathsf{Type}}^{\,i}}{\Delta \vdash \overline{\mathsf{val}\ l_i : T_i}^{\,i}}\ \text{SG-Sig}$$

Figure 6.6. Other rules of **Sig**

We check if T_2 satisfies such restriction when the argument type T_1 is a trait in rules ST-Lam and ST-Pi, i.e., rules for type-checking functions and function types.

Handling the object case separately and using projection substitution in ST-AppM is due to value restrictions on member access and lack of explicit or implicit type conversion in **Sig**. Otherwise, consider using ST-AppV for the object case. Since only member access on variables is supported, after substitution, all $x.L$ in T_2 will become $V_1.L$ which is not well-formed. Furthermore, there is no type conversion to transform $V_1.L$ into V. We directly replace all $x.L$ with V, without the need of an individual type conversion rule or operator.

Objects and Traits. Objects and traits are type-checked by ST-Mod and ST-Sig, respectively. The actual type member V is limited to a value. The reason is that an object will be mapped to a dependent sum in λI_Σ and the actual type member corresponds to the first component, which is required to be a value (see SP-Pair). The type of object bindings M is obtained by replacing all abstract type member L with the actual one V. And we use the trait binding S from the annotation T_1 to check if the type of M is $S[L \mapsto V]$. We also require T, the type of V, not to be a trait. Otherwise, there could exists the form $L.l_i$ in object bindings M. Then $M[L \mapsto V]$ contains $V.l_i$, which is an ill-formed member access. Correspondingly, in the typing rule of traits ST-Sig, the type T of abstract type member L should not be a trait.

Member Access. ST-ProjT and ST-Proj are typing rules for type and value member access, respectively. For a certain value member l, its type T_2 is extracted from S by its label and may contain the abstract type member L. We replace all L in T_2 with member access $x.L$ to prevent L being out of scope. **Sig** only supports object member access of variables. Because member access will be translated to projection and opening operations in λI_Σ. Both projection and opening have value restrictions which only take values as input. Variables are values both in λI_Σ and **Sig**,

which satisfies the restriction. Member access on other values, such as objects is not allowed. Otherwise, type conversion will be needed. If $V.l$ is allowed in ST-Proj where V is an object, its type would be $T_2[x \mapsto V.L]$ which needs to further convert the type $V.L$ to the actual type member.

No Subsumption Rule or Subtyping Relation. Finally, for simplicity reasons, we exclude the subsumption rule and subtyping relation in **Sig**. The subsumption rule would make the system *declarative* and harden the development of metatheory. Moreover, we focus on presenting the *typing* of **Sig**, as well as the type-directed elaboration which involves the inferences of casts (see Section 6.4.3). The elaboration of subtyping is less interesting because subtyping relations between traits can be directly mapped to record subtyping in λI_{Σ}. We leave adding subtyping in **Sig** as future work (see Section 8.2).

6.4.3 Elaboration Semantics

The semantics of **Sig** is given by elaboration to λI_{Σ}. The elaboration is a *type-directed* translation of the surface language **Sig** to the target language. For simplicity reasons, we assume the target language is λI_{Σ} extended with *multi-field records*. Records can be encoded using top types with standard techniques [Pierce 2002]. Typing and subtyping rules of records are admissible in λI_{Σ}.

The main purpose of presenting the elaboration semantics is to illustrate how necessary type-safe casts can be automatically added to target terms. The translation process of **Sig** is particularly for the inference of casts with traits and objects. For encoding other constructs with casts, we may need different elaboration approaches. For example, in Section 3.2.1 we use Scott-encodings for elaborating algebraic datatypes of **Fun** into PITS terms with casts.

The judgment for term translation is denoted by $\Delta \vdash E : T \rightsquigarrow e$, shown in Figure 6.7 and 6.8. If we ignore the target term e and trait binding $x = \langle x_1, x_2 \rangle : T$ in Δ, the translation judgment becomes exactly the typing judgment $\Delta \vdash E : T$ (see Figure 6.5). Most translation rules are straightforward mappings to λI_{Σ} terms. We distinguish the translation for terms with or without objects/traits. The name of rules related to traits has a suffix letter "M". We focus on such rules in the following text.

Other translation judgments are shown in Figure 6.9, which reuse the term translation judgment. The context translation judgment $\vdash \Delta \rightsquigarrow \Gamma$ translates a surface context to a target one. In rule TRW-Mod, the trait binding $x = \langle x_1, x_2 \rangle : T$ is used to save the translation result of an *object variable* x. We decompose x as fresh variables x_1 and x_2, which represent the first and second components of a dependent sum translated from an object. The object and trait binding translation judgments, i.e., $\Delta \vdash M : S \rightsquigarrow e$ and $\Delta \vdash S \rightsquigarrow e$, translate bindings in objects and traits to records and record types, respectively.

Object Variables. Rule TR-VarM in Figure 6.7 translates a variable x when it is a trait binding $x = \langle x_1, x_2 \rangle : T$ from the context. Its type T is a trait and translated into a Sigma-type $\Sigma x_1 : A.\ B$. The object variable x is then translated to a dependent sum $\langle x_1, x_2 \text{ as } \Sigma x_1 : A.\ B \rangle$. Note that x does not show in the translated term. One can only refer to x by its component variables x_1 or x_2, which are both already in the translated context by TRW-Mod. This rule follows the form of the translation of object definitions in rule TR-Mod. Thus, we can translate an object value V in a unified form $\Delta \vdash V : T \rightsquigarrow \langle e_1, e_2 \text{ as } \Sigma x : A.\ B \rangle$.

$\boxed{\Delta \vdash E : T \rightsquigarrow e}$ *(Term Translation)*

$$\textsc{TR-Ax}\ \frac{\vdash \Delta \rightsquigarrow \Gamma}{\Delta \vdash \mathsf{Type} : \mathsf{Type} \rightsquigarrow \star} \qquad \textsc{TR-Var}\ \frac{x : T \in \Delta \qquad \vdash \Delta \rightsquigarrow \Gamma}{\Delta \vdash x : T \rightsquigarrow x} \qquad \textsc{TR-VarM}\ \frac{x = \langle x_1, x_2 \rangle : T \in \Delta \qquad \vdash \Delta \rightsquigarrow \Gamma \qquad \Delta \vdash T : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B}{\Delta \vdash x : T \rightsquigarrow \langle x_1, x_2\ \mathsf{as}\ \Sigma x_1 : A.\ B \rangle}$$

$$\textsc{TR-Pi}\ \frac{\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow A_1 \qquad \Delta, x : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow A_2 \qquad T_1\ \text{is not a trait}}{\Delta \vdash (x : T_1) \to T_2 : \mathsf{Type} \rightsquigarrow \Pi x : A_1.\ A_2}$$

$$\textsc{TR-Lam}\ \frac{\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow A \qquad \Delta, x : T_1 \vdash E : T_2 \rightsquigarrow e \qquad T_1\ \text{is not a trait}}{\Delta \vdash (\lambda(x : T_1) \Rightarrow E) : (x : T_1) \to T_2 \rightsquigarrow \lambda x : A.\ e}$$

$$\textsc{TR-PiM}\ \frac{\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B \qquad \Delta, x = \langle x_1, x_2 \rangle : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow C \qquad T_1 = \mathbf{trait}\,\{\,\mathsf{type}\ L : T; S\} \qquad T_2\ \text{is}\ x.L\text{-only-projection}}{\Delta \vdash (x : T_1) \to T_2 : \mathsf{Type} \rightsquigarrow \Pi y : (\Sigma x_1 : A.\ B).\ (\lambda x_1 : A.\ C)\,(y.1)}$$

$$\textsc{TR-LamM}\ \frac{\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B \qquad \Delta, x = \langle x_1, x_2 \rangle : T_1 \vdash E : T_2 \rightsquigarrow e \qquad T_1 = \mathbf{trait}\,\{\,\mathsf{type}\ L : T; S\} \qquad T_2\ \text{is}\ x.L\text{-only-projection}}{\Delta \vdash (\lambda(x : T_1) \Rightarrow E) : (x : T_1) \to T_2 \rightsquigarrow \lambda y : (\Sigma x_1 : A.\ B).\ \mathbf{open}\ y\ \mathbf{as}\ \langle x_1, x_2 \rangle\ \mathbf{in}\ e}$$

Figure 6.7. Translation of terms

Functions with Traits. Rules TR-PiM and TR-LamM in Figure 6.7 translate function types and functions where the argument type T_1 is a trait. The trait T_1 is translated to a Sigma-type $\Sigma x_1 : A.\ B$. The formal parameter x is added into the context as a trait binding $x = \langle x_1, x_2 \rangle : T_1$ correspondingly. Recall that in TR-VarM and TRW-Mod, x does not exist in the translated context and can only be referred by component variables x_1 or x_2. In TR-LamM, the function body E will be translated to e. The member access operations in E will be directly mapped to operations on component variables in e (see TR-ProjT and TR-Proj). Such behavior can be mapped to the opening operation in λI_Σ, which replaces the input variable x with two pattern variables x_1 and x_2. The whole function $\lambda(x : T_1) \Rightarrow E$ is then translated into a lambda term with opening. We explicitly replace x with a fresh variable y in the translated term to distinguish from x_1 and x_2. Rule TR-PiM matches the type of the translated term in TR-LamM. The whole translated type is a Pi-type, parametrized by the fresh variable y. The body is exactly the type of the **open**-term in TR-LamM, which follows SP-Open of the target language.

Applications with Traits. Functions with traits as input are translated to lambdas with **open**-terms in TR-LamM separately from ordinary functions. Thus, applications with such functions also need a special treatment. Rules TR-AppX and TR-AppM in Figure 6.8 translate dependent functions applied to object values, which cover two cases: variables x or **obj** definitions, respectively.

$\boxed{\Delta \vdash E : T \rightsquigarrow e}$ *(Term Translation)*

TR-App
$$\frac{\Delta \vdash E_1 : T_1 \to T_2 \rightsquigarrow e_1 \quad \Delta \vdash E_2 : T_1 \rightsquigarrow e_2 \qquad T_1 \text{ is not a trait}}{\Delta \vdash E_1\ E_2 : T_2 \rightsquigarrow e_1\ e_2}$$

TR-AppV
$$\frac{\Delta \vdash E : (x : T_1) \to T_2 \rightsquigarrow e_1 \quad \Delta \vdash V : T_1 \rightsquigarrow e_2 \qquad T_1 \text{ is not a trait}}{\Delta \vdash E\ V : T_2[x \mapsto V] \rightsquigarrow e_1\ e_2}$$

TR-AppX
$$\frac{\Delta \vdash E : (x : T_1) \to T_2 \rightsquigarrow e \quad \Delta \vdash y : T_1 \rightsquigarrow \langle y_1, y_2 \text{ as } \Sigma x_1 : A.\ B \rangle \quad T_1 = \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\}}{\Delta \vdash E\ y : T_2[x \mapsto y] \rightsquigarrow \mathsf{cast}^2_{\downarrow}\ (e\ \langle y_1, y_2 \text{ as } \Sigma x_1 : A.\ B \rangle)}$$

TR-AppM
$$\frac{\Delta \vdash E : (x : T_1) \to T_2 \rightsquigarrow e \quad \Delta \vdash V_1 : T_1 \rightsquigarrow \langle e_1, e_2 \text{ as } \Sigma x_1 : A.\ B \rangle \quad T_1 = \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} \quad V_1 = \mathbf{obj}\,\{\,\mathsf{type}\,L = V; M\}\ \mathbf{as}\ T_1}{\Delta \vdash E\ V_1 : T_2[x.L \mapsto V] \rightsquigarrow \mathsf{cast}^2_{\downarrow}\ (e\ \langle e_1, e_2 \text{ as } \Sigma x_1 : A.\ B \rangle)}$$

TR-Mod
$$\frac{\Delta \vdash V : T \rightsquigarrow e_1 \quad \Delta \vdash M[L \mapsto V] : S[L \mapsto V] \rightsquigarrow e_2 \quad \Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B \quad T_1 = \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} \qquad T \text{ is not a trait}}{\Delta \vdash \mathbf{obj}\,\{\,\mathsf{type}\,L = V; M\}\ \mathbf{as}\ T_1 : \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} \rightsquigarrow \langle e_1, e_2 \text{ as } \Sigma x_1 : A.\ B \rangle}$$

TR-Sig
$$\frac{\Delta \vdash T : \mathsf{Type} \rightsquigarrow A \quad \Delta, L : T \vdash S \rightsquigarrow B \qquad T \text{ is not a trait}}{\Delta \vdash \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B[L \mapsto x_1]}$$

TR-ProjT
$$\frac{\Delta \vdash x : T_1 \rightsquigarrow \langle x_1, x_2 \text{ as } \Sigma x_1 : A.\ B \rangle \quad T_1 = \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\}}{\Delta \vdash x.L : T \rightsquigarrow x_1}$$

TR-Proj
$$\frac{\Delta \vdash x : T_1 \rightsquigarrow \langle x_1, x_2 \text{ as } \Sigma x_1 : A.\ B \rangle \quad T_1 = \mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\} \qquad \mathsf{val}\,l : T_2 \in S}{\Delta \vdash x.l : T_2[L \mapsto x.L] \rightsquigarrow x_2.l}$$

Figure 6.8. Translation of terms (cont.)

Note that the type of a translated trait function, i.e., a **open**-term, is a type-level application $(\lambda x_1 : A.\ C)\,(y.1)$ in TR-PiM. We need two consecutive $\mathsf{cast}_{\downarrow}$s in the target term to reduce the first projection and lambda application, similarly to the treatment of second projections (see Section 6.2.3). The function part E is translated to e by TR-LamM. The argument part is translated to a dependent sum. In TR-AppX, the object variable is translated to a dependent sum with component variables by TR-VarM. In TR-AppM, the object definition V_1 is translated to a dependent sum by TR-Mod.

Other Translations with Traits. TR-Mod translates an object definition into a dependent sum. The actual type member V is mapped to the first component e_1. The object bindings M are mapped to the second component e_2, which is a record by object binding translation $\Delta \vdash M : S \rightsquigarrow e$. The members in M have the same labels as in e_1, which can be extracted directly by record projections. TR-Sig translates a trait into a Sigma-type. The type of abstract type member T is translated to the binder type A. The trait binding S is translated to a record

$\boxed{\vdash \Delta \rightsquigarrow \Gamma}$ *(Context Translation)*

$$\textsc{TRW-Nil}\ \frac{}{\vdash \varnothing \rightsquigarrow \varnothing} \qquad \textsc{TRW-Cons}\ \frac{\vdash \Delta \rightsquigarrow \Gamma \quad \Delta \vdash T : \mathsf{Type} \rightsquigarrow A \quad x \text{ fresh in } \Delta}{\vdash \Delta, x : T \rightsquigarrow \Gamma, x : A} \qquad \textsc{TRW-Mod}\ \frac{\vdash \Delta \rightsquigarrow \Gamma \quad \Delta \vdash T : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B \quad x \text{ fresh in } \Delta \quad x_1, x_2 \text{ fresh in } \Gamma}{\vdash \Delta, x = \langle x_1, x_2 \rangle : T \rightsquigarrow \Gamma, x_1 : A, x_2 : B}$$

$\boxed{\Delta \vdash M : S \rightsquigarrow e}$ *(Object Binding Translation)*

$$\textsc{TRM-Bind}\ \frac{\overline{\Delta \vdash E_i : T_i \rightsquigarrow e_i}^{\,i}}{\Delta \vdash \overline{\mathsf{val}\ l_i = E_i}^{\,i} : \overline{\mathsf{val}\ l_i : T_i}^{\,i} \rightsquigarrow \{\, \overline{l_i = e_i}^{\,i} \,\}}$$

$\boxed{\Delta \vdash S \rightsquigarrow e}$ *(Trait Binding Translation)*

$$\textsc{TRS-Sig}\ \frac{\overline{\Delta \vdash T_i : \mathsf{Type} \rightsquigarrow A_i}^{\,i}}{\Delta \vdash \overline{\mathsf{val}\ l_i : T_i}^{\,i} \rightsquigarrow \{\, \overline{l_i : A_i}^{\,i} \,\}}$$

Figure 6.9. Translation of contexts and bindings

type B by trait binding translation $\Delta \vdash S \rightsquigarrow e$. The type and value member access operations are translated by TR-ProjT and TR-Proj, respectively. Type member access is directly mapped to the first component. Value member access is mapped to record projection of the second component.

6.4.4 Soundness of Translation

We show that the type-direct translation of **Sig** is sound, i.e., the translated target terms are well-typed. The main lemma is stated as follows:

Lemma 6.4.1 (Soundness of Translation ✎)**.**

Given $\vdash \Delta \rightsquigarrow \Gamma$*, if* $\Delta \vdash E : T \rightsquigarrow e$ *and* $\Delta \vdash T : \mathsf{Type} \rightsquigarrow A$*, then* $\Gamma \vdash e : A$*.*

The proof is straightforward by induction on the derivation of the term translation judgment $\Delta \vdash E : T \rightsquigarrow e$.

Substitution Lemmas. The soundness proof depends on special substitution lemmas. As mentioned in Section 6.4.2, rule ST-AppM involves the projection substitution with form $T_2[x.L \mapsto V]$ that replaces a type member access with a value. Also, value substitution such as $E[x \mapsto V]$ for an arbitrary term E may fail since E may contain member access operations $x.L$ and $x.l$, which are only for variables and become malformed after substitution with V. If terms do not have member access on variables to substitute, the terms should keep well-formedness after substitution. We define such terms formally as *non-projection terms*:

Definition 6.4.2 (Non-projection Terms)**.** *We call* E x-non-projection *if it does not contain any projection form of* x *(i.e.* $x.L$ *or* $x.l$ *where* x *occurs free in* E*). Similarly, bindings* Δ*,* M *and* S *are* x*-non-projection if they only contain* x*-non-projection terms.*

It is easy to see that all sub-terms of a x-non-projection term are still x-non-projection. Thus, a substitution "$x \mapsto V$" on x-non-projection terms is safe. We can show the following value, typing and projection substitution lemmas:

Lemma 6.4.2 (Value Substitution ✎). *If V_1 is a x-non-projection value and V_2 is a value, then $V_1[x \mapsto V_2]$ is still a value.*

Lemma 6.4.3 (Typing Substitution ✎). *If $\Delta_1, z : T_1, \Delta_2 \vdash E : T \rightsquigarrow e$ and $\Delta_1 \vdash V_2 : T_1 \rightsquigarrow v$, where E and Δ_2 are z-non-projection, then $\Delta_1, \Delta_2[z \mapsto V_2] \vdash E[z \mapsto V_2] : T[z \mapsto V_2] \rightsquigarrow e[z \mapsto v]$.*

Lemma 6.4.4 (Projection Substitution ✎). *Given $T' = \mathbf{trait}\,\{\,\mathsf{type}\,L_1 : T_3; S_1\}$, if $\Delta_1, z = \langle z_1, z_2\rangle : T', \Delta_2 \vdash E : T \rightsquigarrow e$ and $\Delta_1 \vdash V_2 : T_3 \rightsquigarrow v$, where E and Δ_2 are $z.L_1$-only-projection, then $\Delta_1, \Delta_2[z.L_1 \mapsto V_2] \vdash E[z.L_1 \mapsto V_2] : T[z.L_1 \mapsto V_2] \rightsquigarrow e[z_1 \mapsto v]$.*

Proofs are straightforward by induction. Notice that for value substitution lemma, when V_1 is a type member access, it cannot be $x.L$ since it is x-non-projection. Thus, $V_1 = y.L$ for some $y \neq x$, then $V_1[x \mapsto V_2] = V_1 = y.L$ is still a value.

CHAPTER 7

Related Work

7.1 Dependently Typed Calculi without Subtyping

7.1.1 Core Calculus for Functional Languages

Girard's System F_ω [Girard 1972] is a typed lambda calculus with higher-kinded polymorphism. For the well-formedness of type expressions, an extra level of *kinds* is added to the system. In comparison, because of unified syntax, PITS is considerably simpler than System F_ω, both in terms of language constructs and complexity of proofs. As for type-level computation, System F_ω differs from 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 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 and strong sums, which seem hard to support in FC and its extensions. For example, we have presented the $\lambda I_{\leqslant}$ calculus that extends a variant of PITS (i.e. λI) with subtyping in Chapter 5, and the λI_{Σ} calculus which is a variant of $\lambda I_{\leqslant}$ with strong sums in Chapter 6.

7.1.2 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 and Meijer 1997]. The Henk proposal was to use the *lambda cube* as a typed intermediate language, unifying all three levels, i.e., terms, types and

kinds. 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 and 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.

7.1.3 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. However, 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 1986b] also features general recursion to implement equi-recursive types. Recursion and recursive types are unified in a single construct. However, both equi-recursive types and the Type:Type axiom make the type system undecidable. $\Pi\Sigma$ [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, $\Pi\Sigma$ does not have decidable type checking due to the "type-in-type" axiom, and its metatheory is not formally developed.

7.1.4 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 and 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 and Weirich 2015], which has no such separation, employs a 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.

Table 7.1. Comparison between $\lambda I_{\leqslant}$ and related calculi

Features	$\lambda I_{\leqslant}$	$F_{\leqslant}$	PTS$^{\leqslant}$	PSS	$\lambda P_{\leqslant}$	$\lambda\Pi_{\leqslant}$	$\lambda C_{\leqslant}$
Dependent types	●	○	●	●	●	●	●
Beta equality	○	○	●	●	●	●	●
Unified syntax	●	○	●	●	○	○	●
Contravariance	●	●	●	○	●	●	●
Bounded quantification	●	●	◐[1]	●	○	○	○
Top type	●	●	○	●	○	○	○
Transitivity not rely on normalization	●	●	●	—[2]	○	○	●

[1] No subtyping relation for lambdas with bounded quantification.
[2] Metatheory not fully developed.

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.

7.1.5 Restricted Recursion with Termination Checking

As proof assistants, dependently typed languages such as Coq [The Coq development team 2016] and Agda [Norell 2007b] 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.

7.2 Calculi with Subtyping and Dependent Types

In this section, we discuss several related calculi that support subtyping and dependent types (except System $F^{\omega}_{\leqslant}$ and $F^{\omega}_{::}$). The comparison of features between several closely related calculi and $\lambda I_{\leqslant}$ is summarized in Table 7.1. One quick observation from the table is that only $\lambda I_{\leqslant}$ can support all language features of $F_{\leqslant}$, while other calculi cannot fully subsume $F_{\leqslant}$.

7.2.1 Subtyping with Unified Syntax

It is appealing to combine subtyping with the unified syntax of Pure Type Systems [Barendregt 1991] (PTS) for obtaining a concise and expressive system. Chen proposed $\lambda C_{\leqslant}$ [Chen 1997], an extension of the calculus of constructions (λC) with subtyping. $\lambda C_{\leqslant}$ supports neither top types nor bounded quantification in order to simplify the metatheory. The proof of transitivity in $\lambda C_{\leqslant}$ is simpler and does not depend on strong normalization, though decidability still depends on

strong normalization as in λC. Zwanenburg proposed $PTS^{\leqslant}$ [Zwanenburg 1999] by extending PTS with subtyping and bounded quantification. It has the PTS-style unified syntax but with two distinct forms of abstraction for type and bound. In $PTS^{\leqslant}$, the subtyping rules do not depend on the typing rules, which allows proving subtyping properties independently from typing properties. However, such design makes it difficult to extend the framework with two desirable features: 1) subtyping on bounded abstractions, since subtyping rules are defined only for pre-terms; 2) top types, since the subtyping rule of top types depends on typing. Neither of those features are supported by $PTS^{\leqslant}$.

Hutchins proposed another framework called *Pure Subtype Systems* [Hutchins 2010] (PSS) which also adopts the unified syntax based on PTS. The design is simplified by making the system solely based on subtyping without the typing relation. The simplicity of the system comes at the cost of the complexity of metatheory. The proof of transitivity elimination is partial, and therefore subject reduction cannot be proved. Note that although $\lambda I_{\leqslant}$ shares the similar idea of being based on the subtyping relation, it has two major differences from PSS. First, $\lambda I_{\leqslant}$ unifies subtyping with typing in a more conservative way. The unified subtyping relation still tracks types and it intuitively subsumes the traditional typing relation. In contrast, PSS takes a more aggressive approach to make the typing relation completely absent from the system. In PSS there are no types or typing. Second, PSS eliminates the distinction of function and function types, which are unified into the same syntax of abstraction. In contrast, $\lambda I_{\leqslant}$ still distinguishes these two concepts as in PTSs. Since the subtyping rule of abstractions in PSS is pointwise, any form of contravariance is not supported. An unfortunate consequence is that PSS cannot subsume System $F_{\leqslant}$ with contravariant arrow types, including the Kernel Fun variant [Cardelli and Wegner 1985].

7.2.2 Stratified Syntax with High-Order Subtyping

System $F^{\omega}_{\leqslant}$ is a lambda calculus with stratified syntax by extending System F_{ω} [Girard 1972] with higher-order subtyping. To simplify the metatheory, early formalizations of System $F^{\omega}_{\leqslant}$ [Pierce and Steffen 1997; Compagnoni 1995] do not allow a bounded type operator. Compagnoni and Goguen later proposed a technique called typed operational semantics [Compagnoni and Goguen 2003] to fully enable bounded quantification in System $F^{\omega}_{\leqslant}$. But its metatheory becomes quite complicated and relies on strong normalization, making it hard to apply such technique to systems with general recursion. Note that Compagnoni and Goguen's presentation of System $F^{\omega}_{\leqslant}$ contains a kinded subtyping judgment $\Gamma \vdash A \leqslant B : K$ which has a similar shape to the unified subtyping relation in $\lambda I_{\leqslant}$. But the typing relation is separately defined in their system and not subsumed by the kinded subtyping judgment. System $F^{\omega}_{::}$ [Stucki 2017] is an extension of $F^{\omega}_{\leqslant}$ with *type intervals*. $F^{\omega}_{::}$ supports higher-order subtyping with both lower and upper bounded quantifications but its metatheory is complex due to the stratified syntax. The canonical/algorithmic system of $F^{\omega}_{::}$ relies on normalization. The transitivity rule cannot be fully eliminated but partially eliminated for top-level uses.

7.2.3 Stratified Subtyping Systems with Dependent Types

System $\lambda P_{\leqslant}$ [Aspinall and Compagnoni 1996] is a stratified system with dependent types and higher-order subtyping. The metatheory becomes more complex than System $F^{\omega}_{\leqslant}$ due to the circular dependency of typing, kinding and subtyping. A novel proof technique that splits beta reduction on terms and types is proposed to break such dependency. However, System $\lambda P_{\leqslant}$ does

not support polymorphism (i.e. abstraction over types), bounded quantification or top types. System $\lambda\Pi_{\leqslant}$ [Castagna and Chen 2001; Chen 1998] is an improvement of $\lambda P_{\leqslant}$. It has the property of complete type-level transitivity elimination, while System $\lambda P_{\leqslant}$ has transitivity elimination only for normalized types. However, $\lambda\Pi_{\leqslant}$ is proved to be equivalent to $\lambda P_{\leqslant}$ in typing and subtyping, meaning that it has no increased expressiveness. Both $\lambda\Pi_{\leqslant}$ and $\lambda P_{\leqslant}$ require strong normalization to prove the transitivity of subtyping. In contrast, $\lambda I_{\leqslant}$ employs iso-types for explicit type-level computations and decouples strong normalization from the proofs of metatheory.

7.2.4 Subtyping with Restricted Dependent Types

There have been several studies focusing on exploring subtyping with *restricted* forms of dependent types but not *full* dependent types in the context of object-oriented (OO) programming. The *Dependent Object-Oriented Language* [Campos and Vasconcelos 2015, 2018] (DOL) is an imperative OO programming language with subtyping and *index refinements*, a restricted notion of dependent types originated from Dependent ML [Xi and Pfenning 1999], which allows types to depend on static indices of natural numbers. DOL supports the verification of mutable objects and unrestricted use of shared objects. The type checking of DOL is decidable. Compared to $\lambda I_{\leqslant}$, DOL does not support full dependent types or general bounded quantification. In DOL, Pi-types are denoted by $\Pi a : I.T$ and can be quantified only by index types I. Quantifiers in Pi-types only support propositions on indexes, such as comparing natural numbers $a \leqslant b$.

νObj [Odersky et al. 2003] is a dependently typed calculus for objects with type members. It is developed as a theoretic foundation for Scala [Odersky et al. 2004] and features a weaker form of dependent types called *path-dependent types*. In νObj, types can depend on paths which are type selections on variables, i.e., $x.L$. Compared to traditional dependent types used in $\lambda I_{\leqslant}$, it is difficult to use path-dependent types to model dependency on non-path values, e.g., $\Pi n : Int.\ Vec\ n$. The richness of the type system makes the metatheory of νObj complex and type checking is not decidable. Another recent effort of developing a core calculus for Scala is the *Dependent Object Types* (DOT) calculus [Amin et al. 2012b, 2014; Rompf and Amin 2016; Amin et al. 2016]. DOT is also based on path-dependent types. It is simpler and has fewer type forms than νObj, e.g., no class types, but still expressive to model many features of Scala. Similarly to $\lambda I_{\leqslant}$, DOT subsumes System $F_{\leqslant}$ but has a richer notion of bounds. Type variables can be quantified by both lower bounds and upper bounds, as opposed to the traditional bounded quantification used in $\lambda I_{\leqslant}$ that only supports upper bounds. The metatheory of DOT is well-developed [Rompf and Amin 2016], though the soundness proof requires many non-standard techniques. Transitivity of subtyping needs to be treated as an axiom and transitivity elimination is not possible [Rompf and Amin 2016]. Both νObj and DOT use the stratified syntax in contrast to the unified syntax of $\lambda I_{\leqslant}$.

7.3 Strong Sum Types and ML Modules

7.3.1 Dependently Typed Calculi with Strong Sigma-types

It is challenging to combine strong dependent sums with traditional dependently typed systems. If both impredicative polymorphism and strong sums are allowed, Girard's paradox can be derived and logical consistency is lost [Coquand 1986; Hook and Howe 1986]. Previous studies usually drop impredicativity to obtain a consistent system with strong sums. XML [Harper and Mitchell

1993] extends Standard ML with a module system encoded by strong dependent sums. XML uses implicit type conversions by equality relations and utilizes stratified universes to abandon impredicative polymorphism.

Instead of completely dropping impredicativity, another feasible approach [Harper and Mitchell 1993] is to limit predicativity only in strong Sigma-types and allow impredicativity in Pi-types. However, such predicative Sigma-types are more restricted and less expressive, since both the binder type and body should be at the same small kind level, i.e., for $\Sigma x : A.\ B$, both $A : \star$ and $B : \star$ are required. For example, $\Sigma x : \star.\ x$ is not predicative with stratified universes where $\star : \Box$. The Calculus of Dependent Lambda Eliminations (CDLE) [Stump 2017] supports impredicativity and encoding strong sums without treating them as built-in constructs, though the encoded Sigma-types are predicative. Bowman et al. [2017] proposed an extension of the Calculus of Constructions with strong dependent sums for type-preserving CPS translation. Similarly, in their system Pi-types are impredicative but strong sum types are predicative so as to prevent the inconsistency issue.

The λI_{Σ} calculus supports impredicativity both in Pi-types and strong Sigma-types by using a single unified universe (i.e. $\star$) and the "type-in-type" axiom. Thus, the calculus is logically inconsistent. Nonetheless, we do not consider inconsistency problematic for λI_{Σ}. We focus more on the traditional programming patterns, while the related studies mentioned above mostly focus on logical uses such as proof assistant. The loss of strong normalization does not cause problems either, since iso-types untangles normalization from other properties of λI_{Σ}.

7.3.2 Strong Sigma-types with Subtyping

It is even more challenging to allow subtyping with Sigma-types. There is not so much existing work due to the complexity of combining subtyping and dependent sum types. On exception is the work by Luo et al. [2004] on coherent subtyping rules of strong Sigma-types. They propose a new subtyping rule for the first projection of Sigma-types and achieve both coherence and admissibility of transitivity. Their system only supports *predicative* strong sums, similarly to the calculi by Stump [2017] and Bowman et al. [2017]. Several other work, such as dependent records by Pollack [2002], is based on Luo et al.'s rules. One fundamental difference is that they use *coercive subtyping*, while we use structural subtyping based on the approach of unified subtyping [Yang and Oliveira 2017]. Coercive subtyping allows more flexible subtyping rules in their system, however also requires the additional relation, i.e., coercion, and needs to ensure its coherence. The unified subtyping approach used in λI_{Σ} directly reasons on the structures of types.

7.3.3 Core Languages for Scala

As shown in the surface language **Sig**, dependent sums in λI_{Σ} can model several Object-oriented Programming (OOP) structures in Scala [Odersky et al. 2004], e.g., traits and type members. There are several calculi which aim to serve as theoretical foundations of Scala as mentioned in Section 5.6, including νObj [Odersky et al. 2003] and *Dependent Object Types* (DOT) calculus [Amin et al. 2012b, 2014; Rompf and Amin 2016; Amin et al. 2016]. The key feature is *path-dependent types* for modeling member access of traits. νObj supports full paths $p.L$, while DOT supports short paths on variables only, i.e., $x.L$, similarly to the restriction in **Sig**. Both calculi can model more complete Scala features, including both lower and upper bounds for type members, which is missing in λI_{Σ}. Nonetheless, λI_{Σ} is not intended for modeling full Scala features, though it can

support some features of type members as shown in **Sig**. Traditional full dependent types, such as $\Pi n : Int.\ Vec\ n$, are not expressible in νObj or DOT. Higher-kinded types are not currently available in DOT but supported by $F^{\omega}_{::}$ [Stucki 2017], a F_{ω} variant of DOT with upper and lower bounds but without type members or path-dependent types. λI_{Σ} supports both dependent types and high-kinded types. Both transitivity elimination and type-safety are proved for λI_{Σ}. In contrast, νObj does not have fully-developed metatheory. DOT has a relatively more complex soundness proof [Rompf and Amin 2016; Rapoport et al. 2017]. DOT needs a built-in transitivity rule and does not support transitivity elimination.

7.3.4 Encoding ML Modules by Dependent Types

Dependent sums are the key novelty of λI_{Σ} over $\lambda I_{\leqslant}$. Besides OOP language constructs, dependent sums can also model ML-like model systems. MacQueen [1986] proposed such an idea to model functors and signatures using dependent types and *strong* existential types, i.e., strong sum types. Several other work also tries to encode modules with strong sums, e.g., XML by Harper and Mitchell [1993], as well as Leroy's calculus that employs both strong sums and weak manifest sums [Leroy 1994]. For a proper module system, one needs the strong existential opening and second projection from strong sums, because traditional existential opening is too restrictive. It does not allow free access to the witness and interpretation components of an existential package.

Notice that all these previous studies use an implicit conversion rule with equality judgments, while λI_{Σ} employs the iso-type approach with explicit type casts. Instead of using the standard typing rule for the second projection, in λI_{Σ}, we slightly weaken the typing rule of the second projection and strong opening to make them compatible with call-by-value casts. Alternatively, strong sums can be typed using full casts with parallel reduction. But the cost is the complexity of metatheory and the difficulty to combine with other features, such as subtyping.

7.3.5 Encoding ML Modules by F-ing Modules

Instead of directly using dependent sums, ML module systems can be encoded through *indirect semantics* by second-order type systems and ordinary existential types. Rossberg et al. [2010] proposed F-ing modules which elaborate ML modules into System F_{ω}. The target language does not need dependent types or dependent sums. But such encoding requires a complex elaboration process. The semantics of modules is indirect, which is represented differently in the surface and target language. For example, an *abstract* signature is an existential type $\exists\overline{\alpha}.\Sigma$ in the surface ML language that binds abstract types $\overline{\alpha}$. It will be instantiated into a *concrete* signature Σ in F_{ω} which is a record type. Thus, member access of a module can be done internally in an existential package, because the semantics signature is *"virtual"*, i.e., an existential package with abstract binders. There is no actual existential opening and thus the encoding does not require strong existentials, i.e., strong dependent sums.

Type members and traits encoded in **Sig** share some similarities to ML-like modules: traits are like signatures and objects are like structures. We use a type-directed translation from **Sig** to λI_{Σ}, similarly to the elaboration used in F-ing modules. However, our translation process is much simpler. The surface structures of **Sig** have direct mappings in the target language. With dependent sums, we can directly use existential openings. Note that the opening operation in λI_{Σ} has an intermediate type which is a type-level application. The actual type is delayed until the actual implementation is provided, and then can be obtained by $\mathsf{cast}_{\downarrow}$s. This is similar to the

treatment of obtaining concrete signatures in F-ing modules. However, λI_{Σ} uses dependent types and does not need "virtual" signatures that always carry existential binders.

7.3.6 First-class ML Modules

Traditionally, ML modules are treated as a separate system from the core language. Modules cannot be freely mixed with terms. 1ML [Rossberg 2015] is such an attempt to unify the core and modules, which features F-ing modules to elaborate both predicative ML core and module systems into a well-studied target language, i.e., System F_{ω}. Another earlier work by Harper and Lillibridge is *translucent sums* [Harper and Lillibridge 1994; Lillibridge 1997] that support transparent type definitions (i.e. type $L = T$) in signatures. Modules encoded by translucent sums are first-class values. Essentially, translucent sums are weak sums extended with type equality. Similarly to 1ML, Harper and Lillibridge's calculus is based on System F_{ω} but not a dependent type theory.

Thanks to unified syntax in λI_{Σ}, objects of traits are encoded as first-class values in **Sig**. Considering the similarities between traits and modules, **Sig** can also be viewed as a language with lightweight support of first-class ML modules. Though **Sig** lacks many features of ML module systems, such as translucent type members [Leroy 1994; Lillibridge 1997], it still shows the potential of encoding first-class modules with unified syntax and strong dependent sums. Moreover, **Sig** can support signature types that depend on values, e.g., $\mathbf{trait}\ \{\ \mathsf{type}\ L : Int; \mathsf{val}\ l : Vec\ L\}$. Such type is not supported by 1ML or translucent sums, which depend on System F_{ω} that do not have full dependent types.

7.3.7 Module Systems for Dependently Typed Calculi

So far we have discussed module systems for core languages which are non-dependent. Several dependently typed languages, such as Coq [The Coq development team 2016], also employ ML-like module systems. One attempt to formalize such module systems is $\mathcal{MC}_2$ [Courant 2007, 1997] that extends Pure Type Systems with a ML-like module system. The approach of $\mathcal{MC}_2$ is more traditional. Modules in $\mathcal{MC}_2$ are second-class, which are distinct syntactic levels separated from core terms of PTS and require duplicated constructs, such as functors and signatures. In contrast modules/objects in **Sig** are first-class and unified with core terms. Modules/objects are encoded by strong sums but not treated as built-in primitives in λI_{Σ}. Moreover, $\mathcal{MC}_2$ aims at modularizing proofs for proof assistants, while λI_{Σ} and **Sig** focus on modeling modular constructs for traditional programming.

CHAPTER 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we explored the design space of dependently typed languages for general-purpose programming that stand in-between traditional languages and full-spectrum dependently typed languages. We showed how the advantages of dependent types, especially the economy of concepts and added expressiveness, can benefit the designs of traditional languages. In particular, we developed three dependently typed calculi that combine features for traditional programming:

- We developed **Pure Iso-Type Systems** (PITS), a family of dependently typed calculi with general recursion. PITS employs unified syntax and has comparable simplicity to Pure Type Systems (PTS). To retain decidable type checking in the presence of general recursion, we proposed **iso-types**. Iso-types make every type-level computation steps explicit by cast operators and decouple properties such as decidability of type checking from strong normalization. We studied three variants of PITS that differ on reduction strategies of casts and have trade-offs in terms of expressiveness and simplicity of metatheory. We proved type-safety and decidability of type checking for all variants.
- We developed **the $\lambda I_{\leqslant}$ calculus**, a dependently typed calculus with subtyping. $\lambda I_{\leqslant}$ is a variant of PITS with extra features for object-oriented programming, including higher-order subtyping, bounded quantification and top types. To address the issues arising from combining dependent types and subtyping, we proposed **unified subtyping**. Unified subtyping combines typing and subtyping into a single relation and eliminates the circularity of typing and subtyping. We developed the metatheory of $\lambda I_{\leqslant}$ and proved transitivity of subtyping and type-safety. We also showed that $\lambda I_{\leqslant}$ can fully subsume System $F_{\leqslant}$.
- We developed **the λI_{Σ} calculus**, a dependently typed calculus with subtyping and strong dependent sums. λI_{Σ} employs the approach of unified subtyping and is a variant of $\lambda I_{\leqslant}$. λI_{Σ} also features **iso-strong sums** whose destructors are typed as intermediate type-level applications instead of direct substitutions. Call-by-value casts are capable of performing necessary type-level computation without the need of full casts. We proved transitivity and type-safety for λI_{Σ} in the presence of impredicativity. We also showed an application of strong sums for encoding Scala-like traits by elaborating surface constructs of **Sig** into λI_{Σ} terms.

8.2 Future Work

In this section, we discuss several interesting avenues for future work.

Modeling Functional Languages with PITS. We believe that PITS is suitable to serve as a core language for functional languages with expressiveness in-between traditional languages and full-spectrum dependently typed languages. For future work, we would like to employ PITS for modeling 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 have already shown such possibility in Section 3.2 that PITS can model many type-level features which are language extensions to classic Haskell or ML. However, writing a new compiler for full Haskell or ML requires a lot of engineering efforts, even if only considering the classic Haskell 98. We would like to take a more pragmatic approach by modifying existing compilers and replacing the core language with PITS. For example, JHC [Meacham 2006] is a Haskell compiler that implements Haskell 98 and has a much smaller code base than the de facto standard Haskell compiler GHC [The GHC Team 2018]. JHC uses the PTS-style core language Henk [Peyton Jones and Meijer 1997] for implementing type classes. Based on the work of JHC, we hope to replace Henk with PITS and implement more language extensions from modern Haskell such as datatype promotion [Yorgey et al. 2012].

Relax the Value Restriction for Call-by-value PITS. We employ a value restriction in call-by-value PITS to retain subject reduction with a simple proof. As discussed in Section 4.2.1, we will consider an alternative approach by adding cast operators during dynamic semantics for future work. The proposal is to add two special constructs. The syntax of expressions and values is extended as follows:

$$\begin{array}{lcl} e, A & ::= & \cdots \mid (e_1; e_2) \mid \mathsf{cast}_{\uparrow}\ [e_3]\ (e_1; e_2) \\ v & ::= & \cdots \mid (v_1; v_2) \mid \mathsf{cast}_{\uparrow}\ [e_3]\ (e_1; e_2) \end{array}$$

We use the syntax $(e_1; e_2)$ to denote dependent function applications whose arguments are not values. A special cast-up operator denoted by $\mathsf{cast}_{\uparrow}\ [e_3]\ (e_1; e_2)$ is used to expand types of such applications. The typing rules of new constructs are as follows:

$$\frac{\Gamma \vdash e_1 : \Pi x : A.\ B \qquad \Gamma \vdash e_2 : A \qquad x \in \mathsf{FV}(B)}{\Gamma \vdash (e_1; e_2) : (\lambda x : A.\ B)\ e_2}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : A.\ B \qquad \Gamma \vdash e_2 : A \qquad \Gamma \vdash e_3 : A \qquad e_3 \hookrightarrow e_2 \qquad x \in \mathsf{FV}(B)}{\Gamma \vdash \mathsf{cast}_{\uparrow}\ [e_3]\ (e_1; e_2) : (\lambda x : A.\ B)\ e_3}$$

We use the same approach as iso-strong sums (see Section 6.1.2) for the typing of dependent applications. The typing result is a type-level application that carries the argument e_2. The type annotation of the special cast-up operator saves an argument e_3 which is a beta-expansion of the original argument e_2. The special cast-up operator makes it possible to track the types through

reductions. We have the following new reduction rules:

$$\frac{e_1 \hookrightarrow e_1'}{(e_1; e_2) \hookrightarrow (e_1'; e_2)} \qquad \frac{e \hookrightarrow e'}{(v; e) \hookrightarrow \mathsf{cast}_{\uparrow}\ [e]\ (v; e')}$$

$$\frac{}{\mathsf{cast}_{\downarrow}\ (\mathsf{cast}_{\uparrow}\ [e_3]\ (e_1; e_2)) \hookrightarrow (e_1; e_2)} \qquad \frac{}{\mathsf{cast}_{\downarrow}\ (v_1; v_2) \hookrightarrow v_1\ v_2}$$

When trying to reduce $(v; e)$, we will add a cast-up operator to save the original argument and retain the type for the reduced term: $\mathsf{cast}_{\uparrow}\ [e]\ (v; e')$. We can use $\mathsf{cast}_{\downarrow}$ operators to eliminate the cast-up operators and the special form of dependent applications. Our preliminary experiment shows that with this approach we can prove type-safety, namely subject reduction and progress lemmas. Although the typing and reduction rules become complex, we hope that this approach will help recover some missing expressiveness due to the value restriction.

Push Rules for Full PITS. In Section 4.3.2, we discuss an alternative design of full PITS that we can use "push rules" as in System FC [Sulzmann et al. 2007] instead of erasing all casts like Zombie [Sjöberg and Weirich 2015] or Guru [Stump et al. 2008]. For example, we can define a push rule for the case when the function is wrapped by a $\mathsf{cast}_{\Uparrow}$ operator:

$$\frac{}{(\mathsf{cast}_{\Downarrow}\ [A' \to B']\ (\lambda x : A.\ y))\ e \hookrightarrow \mathsf{cast}_{\Downarrow}\ [B']\ ((\lambda x : A.\ y)\ (\mathsf{cast}_{\Uparrow}\ [A]\ e))}$$

The example from Section 4.3.2 can be reduced as follows:

$$\begin{array}{ll} & (\mathsf{cast}_{\Downarrow}\ [Int \to Int]\ (\lambda x : Id\ Int.\ x))\ 3 \\ \hookrightarrow & \mathsf{cast}_{\Downarrow}\ [Int]\ ((\lambda x : Id\ Int.\ x)\ (\mathsf{cast}_{\Uparrow}\ [Id\ Int]\ 3)) \\ \hookrightarrow & \mathsf{cast}_{\Downarrow}\ [Int]\ (\mathsf{cast}_{\Uparrow}\ [Id\ Int]\ 3) \\ \hookrightarrow & 3 \end{array}$$

where $Id = \lambda y : \star.\ y$. However, this rule is still limited since it only works when the function is non-dependent, i.e., with an arrow type and needs to be exactly a lambda term. For future work, we hope to work out more push rules that can cover all possible cases, so that we can directly prove type-safety with such rules. We also believe that a direct operational semantics of full PITS will be helpful to simplify the metatheory and enable the combination of other techniques, such as unified subtyping.

Consistent Full Reduction for Casts. Full PITS employs two different reduction strategies: a parallel reduction for casts and a call-by-name reduction for term evaluation (see Section 4.3.1). This causes some inconsistency in the calculus, e.g. $\lambda x : Int.\ 1 + 1$ is a value but reducible by full casts, and complicates the design. For future work, we would like to explore a consistent full reduction for both type casts and term evaluation. One possible approach is to use the one-step full beta reduction rules (as in Figure 4.9) with a particular order. We have already proved that the current parallel reduction used by full casts has the same expressive power as full beta reduction (see Lemma 4.3.3). This alternative design will not reduce the expressiveness of the calculus.

Surface Mechanisms for Iso-Types. We would like to explore mechanisms in surface languages that make iso-types more convenient to use. One possible direction is the *lightweight* inference of casts. The end users will be not aware of casts at the surface level and the underlying compiler will help generating necessary casts for keeping type-safety. Note that we do not expect a general full inference of casts, which is difficult or even impossible. A more realistic approach is to infer casts for *specific* constructs. For example, we have shown how to automatically generate casts in the target language by an elaboration of surface constructs, such as algebraic datatypes using Scott encodings (Section 3.2.1) and Scala-like traits using strong sums (Section 6.4.3).

Algorithmic Unified Subtyping. The current algorithmic version of $\lambda I_{\leqslant}$ has a notable difference from the declarative system: the typing and subtyping relations are defined separately (see Section 5.4). This causes complexity in developing metatheory for the algorithmic system. For example, if we want to prove the decidability, we need cover both subtyping and typing judgments. The algorithmic system also heavily relies on the erasure of annotations, making it difficult to prove inversion lemmas. For future work, we hope to develop new *algorithmic unified subtyping* judgments for both $\lambda I_{\leqslant}$ and λI_{Σ} calculi. Instead of separated typing and subtyping judgments, there will be only unified subtyping judgments with two directions: the checking judgment $\Gamma \vdash e_1 \leqslant e_2 \Leftarrow A$ and the synthesis judgment $\Gamma \vdash e_1 \leqslant e_2 \Rightarrow A$. We would like to prove the soundness and completeness of the new judgments to the original declarative unified subtyping judgment.

Decidability of Unified Subtyping. We discussed the impact of using the Pi-type rule with full contravariance in $\lambda I_{\leqslant}$, which makes the unified subtyping relation undecidable (see Section 5.6). However, we did not answer whether the calculus is decidable with the Kernel Fun rule. Like System $F_{\leqslant}$, we need to first create an algorithmic presentation of the calculus. In Section 5.4, we presented a sound and complete algorithmic version of $\lambda I_{\leqslant}$ but did not prove its decidability due to the complexity of its judgments. For future work, we would like to develop sound and complete algorithmic versions of $\lambda I_{\leqslant}$ and λI_{Σ} based on the *algorithmic unified subtyping* judgments and prove their decidability.

More Features for Unified Subtyping. We would like to explore more features for the unified subtyping relation. For example, as discussed in Section 5.6, the full subtyping of recursion is not yet supported with unified subtyping due to the pointwise subtyping rules. Similarly, subtyping applications is not complete, e.g., in rules S-App and SP-App, since the arguments are required to be fixed. We hope to remove such restrictions by the polarized subtyping approach [Steffen 1998], which provides a mechanism called polarities for a finer-grained control of covariance and contravariance.

Another example is to extend the bounded quantification with both lower and upper bounds as in DOT [Amin et al. 2012a]. Currently, only upper bound is supported in $\lambda I_{\leqslant}$, which follows the treatment of System $F_{\leqslant}$. However, the metatheory of DOT is significantly more complex than $\lambda I_{\leqslant}$. If only considering to add bounds, we could adopt the method by Stucki [2017] that extends F_{ω} with type intervals (i.e. both upper and lower bounds), which hopefully will not complicate the metatheory too much. We would also like to take the same approach to add bounds for λI_{Σ} that currently drops bound quantification for simplicity reasons. One long-term goal is to add

more features in λI_{Σ}, making it be able to subsume DOT [Amin et al. 2012a] and model more Scala-like programming idioms.

Relax the Value Restriction in λI_{Σ}. Similarly to call-by-value PITS, λI_{Σ} uses call-by-value casts and imposes value restrictions. For future work, we would like to relax value restrictions of λI_{Σ} and introduce a special $\mathsf{cast}_{\uparrow}$ operator, similarly to the previously mentioned proposal for call-by-value PITS:

$$\frac{\Gamma \vdash e : \Sigma x : A.\ B}{\Gamma \vdash e.2 : (\lambda x : A.\ B)\,(e.1)} \qquad \frac{\Gamma \vdash e : \Sigma x : A.\ B \qquad \Gamma \vdash e' : \Sigma x : A.\ B \qquad e \hookrightarrow e'}{\Gamma \vdash \mathsf{cast}_{\uparrow}\,[e]\,(e'.2) : (\lambda x : A.\ B)\,(e.1)}$$

Cast-up operators will be automatically inserted in the reduction rule of the second projections and can also be canceled by $\mathsf{cast}_{\downarrow}$ operators:

$$\frac{e \hookrightarrow e'}{e.2 \hookrightarrow \mathsf{cast}_{\uparrow}\,[e]\,(e'.2)} \qquad \frac{}{\mathsf{cast}_{\downarrow}\,(\mathsf{cast}_{\uparrow}\,[e]\,(e'.2)) \hookrightarrow e'.2}$$

To allow nested second projections, we can use a *full* $\mathsf{cast}_{\Downarrow}$ operator to eliminate the intermediate type-level application:

$$\begin{array}{ll} e & : \Sigma x : A.\ B \\ e.2 & : (\lambda x : A.\ B)\,(e.1) \\ \mathsf{cast}_{\Downarrow}\,(e.2) & : B[x \mapsto e.1] \end{array}$$

Then we can further perform another second projection such as $(\mathsf{cast}_{\Downarrow}\,(e.2)).2$ if B is also a Sigma-type.

Extensions to Sig. The **Sig** language presented in Section 6.4 is a very simple language built directly based on λI_{Σ}. It has several language restrictions to keep the design simple and focus on presenting the application of iso-strong sums. For future work, we would like to support more surface-level features in **Sig**. One important missing feature in **Sig** is subtyping. Since the core language λI_{Σ} already supports (unified) subtyping, we can develop the surface subtyping rules based subtyping rules in λI_{Σ}. For example, the subtyping rule for traits can be defined as follows:

$$\frac{\Delta \vdash T_j \leqslant T'_i \qquad \forall i \in 1..n, \exists j \in 1..m, l_j = l'_i}{\begin{array}{c}\Delta \vdash \mathbf{trait}\,\{\,\mathsf{type}\,L :\ T; \mathsf{val}\,l_1 : T_1;\ ..\ ; \mathsf{val}\,l_m : T_m\} \leqslant \\ \mathbf{trait}\,\{\,\mathsf{type}\,L :\ T; \mathsf{val}\,l'_1 : T'_1;\ ..\ ; \mathsf{val}\,l'_n : T'_n\}\end{array}}$$

This rule follows record subtyping (see Section 2.4.1). Thus, the subtyping relation of traits still holds for the translated terms in the core since traits are translated to Sigma-types with record types.

Another example is to support transparent types, which are useful constructs in the ML module systems. We would like to follow the approach proposed by Leroy [1994] and Lillibridge [1997] to support *translucent sum types*, i.e., dependent sum types containing equality quantifiers, denoted by $\Sigma x = e : A.\ B$. Transparent type definitions in **Sig** can be encoded as follows:

$$\mathbf{trait}\,\{\,\mathsf{type}\,L : T = E\,; \cdots\,\} \rightsquigarrow \Sigma L = e : A.\ \cdots$$

Translucent sums can be modeled as weak sums using weak opening operations [Lillibridge 1997], which will (hopefully) have a relatively simple metatheory.

APPENDIX A

Manual Proofs

A.1 Encoding Weak Sums in $\lambda I_{\leqslant}$

We show the subtyping and typing rules of weak dependent sums are admissible in $\lambda I_{\leqslant}$.

Lemma A.1.1. *The following unified subtyping rule is admissible:*

$$\frac{\Gamma \vdash A_1 \leqslant A_2 : \star \qquad \Gamma, x : A_1 \vdash B_1 \leqslant B_2 : \star}{\Gamma \vdash (\Sigma x : A_1.\ B_1) \leqslant (\Sigma x : A_2.\ B_2) : \star}$$

Proof. By encoding in Section 2.2.1, the conclusion is equivalent to

$$\Gamma \vdash (\Pi z : \star.\ (\Pi x : A_1.\ B_1 \to z) \to z) \leqslant (\Pi z : \star.\ (\Pi x : A_2.\ B_2 \to z) \to z) : \star$$

We show this relation holds:

$\Gamma, z : \star \vdash A_1 \leqslant A_2 : \star$	by Lemma 5.3.4
$\Gamma, z : \star, x : A_1 \vdash B_1 \leqslant B_2 : \star$	by Lemma 5.3.4
$\Gamma, z : \star, x : A_1 \vdash (B_2 \to z) \leqslant (B_1 \to z) : \star$	by rule S-Prod
$\Gamma, z : \star \vdash (\Pi x : A_2.\ B_2 \to z) \leqslant (\Pi x : A_1.\ B_1 \to z) : \star$	by rule S-Prod
$\Gamma, z : \star \vdash ((\Pi x : A_1.\ B_1 \to z) \to z) \leqslant ((\Pi x : A_2.\ B_2 \to z) \to z) : \star$	by rule S-Prod
$\Gamma \vdash (\Pi z : \star.\ (\Pi x : A_1.\ B_1 \to z) \to z) \leqslant (\Pi z : \star.\ (\Pi x : A_2.\ B_2 \to z) \to z) : \star$	by rule S-Prod

Note that $\Gamma, x : A$ is syntactic sugar of $\Gamma, x \leqslant \top : A$. □

Lemma A.1.2. *The typing rules for* **pack** *and* **unpack** *[Schmidt 1994] are admissible.*

Proof. The typing rule for **pack** is

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[x \mapsto e_1]}{\Gamma \vdash \mathbf{pack}\,[e_1, e_2]\,\mathbf{as}\,(\Sigma x : A.\ B) : (\Sigma x : A.\ B)}$$

By encoding in Section 2.2.1, the conclusion is equivalent to

$$\Gamma \vdash (\lambda z : \star.\ \lambda f : (\Pi x : A.\ B \to z).\ f\ e_1\ e_2) : (\Pi z : \star.\ (\Pi x : A.\ B \to z) \to z)$$

We show this relation holds:

$\Gamma, z : \star, f : \Pi x : A.\ B \to z \vdash e_1 : A$	by Lemma 5.3.4
$\Gamma, z : \star, f : \Pi x : A.\ B \to z \vdash e_2 : B[x \mapsto e_1]$	by Lemma 5.3.4
$\Gamma, z : \star, f : \Pi x : A.\ B \to z \vdash f\ e_1 : B[x \mapsto e_1] \to z$	by rule S-App
$\Gamma, z : \star, f : \Pi x : A.\ B \to z \vdash f\ e_1\ e_2 : z$	by rule S-App
$\Gamma \vdash (\lambda z : \star.\ \lambda f : (\Pi x : A.\ B \to z).\ f\ e_1\ e_2) : (\Pi z : \star.\ (\Pi x : A.\ B \to z) \to z)$	by rule S-Abs

The typing rule for **unpack** is

$$\frac{\Gamma \vdash e_1 : (\Sigma x : A.\ B) \qquad \Gamma, x : A, y : B \vdash e_2 : C \qquad \Gamma \vdash C : \star}{\Gamma \vdash \mathbf{unpack}\ e_1\ \mathbf{as}\ [x, y]\ \mathbf{in}\ e_2 : C}$$

By encoding in Section 2.2.1, the typing of e_1 and the conclusion are equivalent to

$$\Gamma \vdash e_1 : (\Pi z : \star.\ (\Pi x : A.\ B \to z) \to z)$$
$$\Gamma \vdash e_1\ C\ (\lambda x : A.\ \lambda y : B.\ e_2) : C$$

where $z \notin \mathsf{FV}(\Pi x : A.\ B)$ and $x, y \notin \mathsf{FV}(C)$. We show the conclusion holds:

$\Gamma \vdash e_1\ C : (\Pi x : A.\ B \to C) \to C$	by rule S-App and z fresh
$\Gamma \vdash (\lambda x : A.\ \lambda y : B.\ e_2) : (\Pi x : A.\ B \to C)$	by rule S-Abs and $y \notin \mathsf{FV}(C)$
$\Gamma \vdash e_1\ C\ (\lambda x : A.\ \lambda y : B.\ e_2) : C$	by rule S-App

□

A.2 Subsumption of System $F_{\leqslant}$ in $\lambda I_{\leqslant}$

Lemma A.2.1 (Commutativity of Type Substitution). $(T_1[X \mapsto T_2])^* = {T_1}^*[X \mapsto {T_2}^*]$ *holds.*

Proof. By induction on the structure of T_1:

- Case $T_1 = \top$:
  $(\top[X \mapsto T_2])^* = \top^* = \top$
  $\top^*[X \mapsto {T_2}^*] = \top[X \mapsto {T_2}^*] = \top$
- Case $T_1 = Y$:
  - Case $X = Y$: $(X[X \mapsto T_2])^* = {T_2}^*$
    $X^*[X \mapsto {T_2}^*] = X[X \mapsto {T_2}^*] = {T_2}^*$
  - Case $X \neq Y$: $(Y[X \mapsto T_2])^* = Y$
    $Y^*[X \mapsto {T_2}^*] = Y[X \mapsto {T_2}^*] = Y$
- Case $T_1 = U_1 \to U_2$:
  IH: $(U_1[X \mapsto T_2])^* = {U_1}^*[X \mapsto {T_2}^*]$
  $(U_2[X \mapsto T_2])^* = {U_2}^*[X \mapsto {T_2}^*]$

  $$\begin{aligned}((U_1 \to U_2)[X \mapsto T_2])^* &= (U_1[X \mapsto T_2] \to U_2[X \mapsto T_2])^* \\ &= \Pi x \leqslant \top : (U_1[X \mapsto T_2])^*.\ (U_2[X \mapsto T_2])^* \quad (x \text{ Fresh}) \\ &= \Pi x \leqslant \top : ({U_1}^*[X \mapsto {T_2}^*]).\ ({U_2}^*[X \mapsto {T_2}^*])\end{aligned}$$

  $$\begin{aligned}(U_1 \to U_2)^*[X \mapsto {T_2}^*] &= (\Pi x \leqslant \top : {U_1}^*.\ {U_2}^*)[X \mapsto {T_2}^*] \quad (x \text{ Fresh}) \\ &= \Pi x \leqslant \top[X \mapsto {T_2}^*] : ({U_1}^*[X \mapsto {T_2}^*]).\ ({U_2}^*[X \mapsto {T_2}^*]) \\ &= \Pi x \leqslant \top : ({U_1}^*[X \mapsto {T_2}^*]).\ ({U_2}^*[X \mapsto {T_2}^*])\end{aligned}$$
- Case $T_1 = \forall Y \leqslant U_1.\ U_2$ and $X \neq Y$:

IH: $(U_1[X \mapsto T_2])^* = U_1{}^*[X \mapsto T_2{}^*]$
$(U_2[X \mapsto T_2])^* = U_2{}^*[X \mapsto T_2{}^*]$

$$\begin{aligned}((\forall Y \leqslant U_1.\ U_2)[X \mapsto T_2])^* &= (\forall Y \leqslant U_1[X \mapsto T_2].\ U_2[X \mapsto T_2])^* \\ &= \Pi Y \leqslant (U_1[X \mapsto T_2])^* : \star.\ (U_2[X \mapsto T_2])^* \\ &= \Pi Y \leqslant (U_1{}^*[X \mapsto T_2{}^*]) : \star.\ (U_2{}^*[X \mapsto T_2{}^*])\end{aligned}$$

$$\begin{aligned}(\forall Y \leqslant U_1.\ U_2)^*[X \mapsto T_2{}^*] &= (\Pi Y \leqslant U_1{}^* : \star.\ U_2{}^*)[X \mapsto T_2{}^*] \\ &= \Pi Y \leqslant (U_1{}^*[X \mapsto T_2{}^*]) : \star[X \mapsto T_2{}^*].\ (U_2{}^*[X \mapsto T_2{}^*]) \\ &= \Pi Y \leqslant (U_1{}^*[X \mapsto T_2{}^*]) : \star.\ (U_2{}^*[X \mapsto T_2{}^*])\end{aligned}$$

□

Lemma A.2.2 (Well-formedness)**.**

1. *If* $\Delta \vdash T$*, then* $\Delta^* \vdash T^* : \star$*.*
2. *If* $\vdash \Delta$*, then* $\vdash \Delta^*$*.*

Proof. By mutual induction on the derivation of $\Delta \vdash T$ and $\vdash \Delta$:

- Case $\dfrac{\vdash \Delta}{\Delta \vdash \top}$:

$\vdash \Delta^*$	by IH
$\Delta^* \vdash \star : \star$	by rule S-Ax
$\Delta^* \vdash \top : \star$	by rule S-TopRefl
$\Delta^* \vdash \top^* : \star$	by definition of T^*

- Case $\dfrac{\vdash \Delta \qquad X \leqslant U \in \Delta}{\Delta \vdash X}$:

$\vdash \Delta^*$	by IH
$X \leqslant U^* : \star \in \Delta^*$	by definition of Δ^*
$\Delta^* \vdash X : \star$	by rule S-VarRefl
$\Delta^* \vdash X^* : \star$	by definition of T^*

- Case $\dfrac{\Delta \vdash U_1 \qquad \Delta \vdash U_2}{\Delta \vdash U_1 \to U_2}$:

$\Delta^* \vdash U_1{}^* : \star$	by IH
$\Delta^* \vdash U_2{}^* : \star$	by IH
$\Delta^* \vdash \top : U_1{}^*$	by rule S-TopRefl
$\Delta^*, x \leqslant \top : U_1{}^* \vdash U_2{}^* : \star$	by Lemma 5.3.4 (x Fresh)
$\Delta^* \vdash \Pi x \leqslant \top : U_1{}^*.\ U_2{}^* : \star$	by rule S-Prod
$\Delta^* \vdash (U_1 \to U_2)^* : \star$	by definition of T^*

- Case $\dfrac{\Delta \vdash U_1 \qquad \Delta, X \leqslant U_1 \vdash U_2}{\Delta \vdash \forall X \leqslant U_1.\ U_2}$:

$\Delta^* \vdash U_1{}^* : \star$	by IH
$(\Delta, X \leqslant U_1)^* \vdash U_2{}^* : \star$	by IH
$\Delta^*, X \leqslant U_1{}^* : \star \vdash U_2{}^* : \star$	by definition of Δ^*
$\Delta^* \vdash \Pi X \leqslant U_1{}^* : \star.\ U_2{}^* : \star$	by rule S-Prod
$\Delta^* \vdash (\forall X \leqslant U_1.\ U_2)^* : \star$	by definition of T^*

- Case $\dfrac{}{\vdash \varnothing}$:

  Trivial.

- Case $\dfrac{\vdash \Delta \qquad \Delta \vdash U}{\vdash \Delta, X \leqslant U}$:

$\vdash \Delta^*$	by IH
$\Delta^* \vdash U^* : \star$	by IH
$\Delta^* \vdash \star : \star$	by rule S-Ax
$\vdash \Delta^*, X \leqslant U^* : \star$	by rule W-Cons
$\vdash (\Delta, X \leqslant U)^*$	by definition of Δ^*

- Case $\dfrac{\vdash \Delta \qquad \Delta \vdash U}{\vdash \Delta, x : U}$:

$\vdash \Delta^*$	by IH
$\Delta^* \vdash U^* : \star$	by IH
$\Delta^* \vdash \top : U^*$	by rule S-TopRefl
$\vdash \Delta^*, x \leqslant \top : U^*$	by rule W-Cons
$\vdash (\Delta, x : U)^*$	by definition of Δ^*

□

Lemma A.2.3 (Subtyping). *If* $\Delta \vdash T_1 \leqslant T_2$, *then* $\Delta^* \vdash {T_1}^* \leqslant {T_2}^* : \star$.

Proof. By induction on the derivation of $\Delta \vdash T_1 \leqslant T_2$:

- Case $\dfrac{\Delta \vdash U}{\Delta \vdash U \leqslant \top}$:

$\Delta^* \vdash U^* : \star$	by Lemma A.2.2
$\Delta^* \vdash U^* \leqslant \top : \star$	by rule S-Top
$\Delta^* \vdash U^* \leqslant \top^* : \star$	by definition of T^*

- Case $\dfrac{\Delta \vdash X}{\Delta \vdash X \leqslant X}$:

$\Delta^* \vdash X^* : \star$	by Lemma A.2.2
$\Delta^* \vdash X^* \leqslant X^* : \star$	i.e.

- Case $\dfrac{X \leqslant U_1 \in \Delta \qquad \Delta \vdash U_1 \leqslant U_2}{\Delta \vdash X \leqslant U_2}$:

$X \leqslant {U_1}^* : \star \in \Delta^*$	by definition of Δ^*
$\Delta^* \vdash {U_1}^* \leqslant {U_2}^* : \star$	by IH
$\Delta^* \vdash X \leqslant {U_2}^* : \star$	by rule S-VarTrans
$\Delta^* \vdash X^* \leqslant {U_2}^* : \star$	i.e.

- Case $\dfrac{\Delta \vdash T_1 \leqslant U_1 \qquad \Delta \vdash U_2 \leqslant T_2}{\Delta \vdash U_1 \to U_2 \leqslant T_1 \to T_2}$:

$\Delta^* \vdash {T_1}^* \leqslant {U_1}^* : \star$	by IH
$\Delta^* \vdash {U_1}^* : \star$	by Lemma 5.3.1
$\Delta^* \vdash {T_1}^* : \star$	by Lemma 5.3.1
$\Delta^* \vdash \top : {T_1}^*$	by rule S-Top
$\Delta^* \vdash {U_2}^* \leqslant {T_2}^* : \star$	by IH
$\Delta^*, x \leqslant \top : {T_1}^* \vdash {U_2}^* \leqslant {T_2}^* : \star$	by Lemma 5.3.4 (x Fresh)
$\Delta^* \vdash \top : {U_1}^*$	by rule S-Top
$\Delta^* \vdash {U_2}^* : \star$	by Lemma 5.3.1
$\Delta^*, x \leqslant \top : {U_1}^* \vdash {U_2}^* : \star$	by Lemma 5.3.4
$\Delta^* \vdash (\Pi x \leqslant \top : {U_1}^*.\ {U_2}^*) \leqslant (\Pi x \leqslant \top : {T_1}^*.\ {T_2}^*) : \star$	by rule S-Prod
$\Delta^* \vdash (U_1 \to U_2)^* \leqslant (T_1 \to T_2)^* : \star$	by definition of T^*

- Case $\dfrac{\Delta, X \leqslant U \vdash T_1 \leqslant T_2}{\Delta \vdash \forall X \leqslant U.\ T_1 \leqslant \forall X \leqslant U.\ T_2}$:

$(\Delta, X \leqslant U)^* \vdash {T_1}^* \leqslant {T_2}^* : \star$	by IH
${\Delta^*}, X \leqslant U^* : \star \vdash {T_1}^* \leqslant {T_2}^* : \star$	by definition of Δ^*
${\Delta^*}, X \leqslant U^* : \star \vdash {T_1}^* : \star$	by Lemma 5.3.1
$\vdash {\Delta^*}, X \leqslant U^* : \star$	by regularity
${\Delta^*} \vdash U^* : \star$	by inversion of rule W-Cons
$\vdash {\Delta^*}$	by regularity
${\Delta^*} \vdash \star : \star$	by rule S-Ax
${\Delta^*} \vdash (\Pi X \leqslant U^* : \star.\ {T_1}^*) \leqslant (\Pi X \leqslant U^* : \star.\ {T_2}^*) : \star$	by rule S-Prod
${\Delta^*} \vdash (\forall X \leqslant U.\ T_1)^* \leqslant (\forall X \leqslant U.\ T_2)^* : \star$	by definition of T^*

□

Lemma A.2.4 (Typing). *If $\Delta \vdash t : T$, then ${\Delta^*} \vdash t^* : T^*$.*

Proof. By induction on the derivation of $\Delta \vdash t : T$:

- Case $\dfrac{\vdash \Delta \qquad x : U \in \Delta}{\Delta \vdash x : U}$:

$\vdash {\Delta^*}$	by Lemma A.2.2
$x \leqslant \top : U^* \in {\Delta^*}$	by definition of Δ^*
${\Delta^*} \vdash x^* : U^*$	by rule S-VarRefl

- Case $\dfrac{\Delta, x : T_1 \vdash t : T_2}{\Delta \vdash \lambda x : T_1.\ t : T_1 \to T_2}$:

$(\Delta, x : T_1)^* \vdash t^* : {T_2}^*$	by IH
${\Delta^*}, x \leqslant \top : {T_1}^* \vdash t^* : {T_2}^*$	by definition of Δ^*
${\Delta^*}, x \leqslant \top : {T_1}^* \vdash {T_2}^* : \star$	by Lemma 5.3.11
$\vdash {\Delta^*}, x \leqslant \top : {T_1}^*$	by regularity
${\Delta^*} \vdash \top : {T_1}^*$	by inversion of rule W-Cons
${\Delta^*} \vdash {T_1}^* : \star$	by inversion of rule W-Cons
${\Delta^*} \vdash (\lambda x \leqslant \top : {T_1}^*.\ t^*) : \Pi x \leqslant \top : {T_1}^*.\ {T_2}^*$	by rule S-Abs
${\Delta^*} \vdash (\lambda x : T_1.\ t)^* : (T_1 \to T_2)^*$	by definition of T^* and t^*

- Case $\dfrac{\Delta \vdash t_1 : U_1 \to U_2 \qquad \Delta \vdash t_2 : U_1}{\Delta \vdash t_1\ t_2 : U_2}$:

${\Delta^*} \vdash {t_1}^* : (U_1 \to U_2)^*$	by IH
${\Delta^*} \vdash {t_1}^* : \Pi x \leqslant \top : {U_1}^*.\ {U_2}^*$	by definition of T^*
${\Delta^*} \vdash {t_2}^* : {U_1}^*$	by IH
${\Delta^*} \vdash {t_2}^* \leqslant \top : {U_1}^*$	by rule S-Top
${\Delta^*} \vdash {t_1}^*\ {t_2}^* : {U_2}^*[x \mapsto {t_2}^*]$	by rule S-App
${U_2}^*[x \mapsto {t_2}^*] = {U_2}^*$	x Fresh
${\Delta^*} \vdash (t_1\ t_2)^* : {U_2}^*$	by definition of t^*

- Case $\dfrac{\Delta, X \leqslant T_1 \vdash t : T_2}{\Delta \vdash \Lambda X \leqslant T_1.\ t : \forall X \leqslant T_1.\ T_2}$:

$(\Delta, X \leqslant T_1)^* \vdash t^* : {T_2}^*$	by IH
$\Delta^*, X \leqslant {T_1}^* : \star \vdash t^* : {T_2}^*$	by definition of Δ^*
$\Delta^*, X \leqslant {T_1}^* : \star \vdash {T_2}^* : \star$	by Lemma 5.3.11
$\vdash \Delta^*, X \leqslant {T_1}^* : \star$	by regularity
$\Delta^* \vdash {T_1}^* : \star$	by inversion of rule W-Cons
$\vdash \Delta^*$	by regularity
$\Delta^* \vdash \star : \star$	by rule S-Ax
$\Delta^* \vdash (\lambda X \leqslant {T_1}^* : \star.\ t^*) : \Pi X \leqslant {T_1}^* : \star.\ {T_2}^*$	by rule S-Abs
$\Delta^* \vdash (\Lambda X \leqslant T_1.\ t)^* : (\forall X \leqslant T_1.\ T_2)^*$	by definition of T^* and t^*

- Case $\dfrac{\Delta \vdash t : \forall X \leqslant U_1.\ U_2 \qquad \Delta \vdash T \leqslant U_1}{\Delta \vdash t[T] : U_2[X \mapsto T]}$:

$\Delta^* \vdash T^* \leqslant {U_1}^* : \star$	by Lemma A.2.3
$\Delta^* \vdash t^* : (\forall X \leqslant U_1.\ U_2)^*$	by IH
$\Delta^* \vdash t^* : \Pi X \leqslant {U_1}^* : \star.\ {U_2}^*$	by definition of T^*
$\Delta^* \vdash t^*\ T^* : {U_2}^*[X \mapsto T^*]$	by rule S-App
${U_2}^*[X \mapsto T^*] = (U_2[X \mapsto T])^*$	by Lemma A.2.1
$\Delta^* \vdash (t[T])^* : (U_2[X \mapsto T])^*$	by definition of t^*

- Case $\dfrac{\Delta \vdash t : T \qquad \Delta \vdash T \leqslant U}{\Delta \vdash t : U}$:

$\Delta^* \vdash t^* : T^*$	by IH
$\Delta^* \vdash T^* \leqslant U^* : \star$	by Lemma A.2.3
$\Delta^* \vdash t^* : U^*$	by rule S-Sub

□

A.3 Soundness of Translation for Sig

Lemma A.3.1 (Translation of Values). *If $\Delta \vdash V : T \rightsquigarrow e$, then e is a value.*

Proof. Trivial by induction on the derivation of $\Delta \vdash V : T \rightsquigarrow e$. □

Lemma A.3.2 (Context Well-formedness). *If $\Delta \vdash E : T \rightsquigarrow e$, then $\vdash \Delta \rightsquigarrow \Gamma$.*

Proof. Trivial by induction on the derivation of $\Delta \vdash E : T \rightsquigarrow e$. □

Lemma A.3.3 (Weakening). *If $\Delta_1, \Delta_3 \vdash E : T \rightsquigarrow e$ and $\vdash \Delta_1, \Delta_2, \Delta_3 \rightsquigarrow \Gamma$, then $\Delta_1, \Delta_2, \Delta_3 \vdash E : T \rightsquigarrow e$.*

Proof. Trivial by induction on the derivation of $\Delta_1, \Delta_3 \vdash E : T \rightsquigarrow e$. □

Lemma A.3.4 (Uniqueness). *If $\Delta \vdash E : T_1 \rightsquigarrow e_1$ and $\Delta \vdash E : T_2 \rightsquigarrow e_2$, then $T_1 = T_2$ and $e_1 = e_2$.*

Proof. Trivial by induction on the derivation of $\Delta \vdash E : T_1 \rightsquigarrow e_1$. □

Definition A.3.1 (Projection-Only Terms). *We call E $x.L$-only-projection if x and L are free variables in E and $x \notin \mathsf{FV}(E[x.L \mapsto y])$ for any fresh variable $y \neq x$. Similarly, bindings Δ, M and S are $x.L$-only-projection if they only contain $x.L$-only-projection terms.*

Lemma A.3.5 (Value Projection Substitution). *If V_1 is a value and V_2 is a value, then $V_1[x.L \mapsto V_2]$ is still a value.*

Proof. Trivial by induction on the form of V_1. □

Lemma A.3.6 (Projection Substitution). *Assume* $T' = \mathbf{trait}\,\{\,\mathsf{type}\,L_1 : T_3; S_1\}$.

(1) If $\Delta_1, z = \langle z_1, z_2\rangle : T', \Delta_2 \vdash E : T \rightsquigarrow e$ *and* $\Delta_1 \vdash V_2 : T_3 \rightsquigarrow v$*, where* E *and* Δ_2 *are* $z.L_1$*-only-projection, then* $\Delta_1, \Delta_2[z.L_1 \mapsto V_2] \vdash E[z.L_1 \mapsto V_2] : T[z.L_1 \mapsto V_2] \rightsquigarrow e[z_1 \mapsto v]$.

(2) If $\vdash \Delta_1, z = \langle z_1, z_2\rangle : T', \Delta_2 \rightsquigarrow \Gamma_1, z_1 : A, z_2 : B, \Gamma_2$ *and* $\Delta_1 \vdash V_2 : T_3 \rightsquigarrow v$*, where* Δ_2 *is* $z.L_1$*-only-projection, then* $\vdash \Delta_1, \Delta_2[z.L_1 \mapsto V_2] \rightsquigarrow \Gamma_1, \Gamma_2[z_1 \mapsto v]$.

(3) If $\Delta_1, z = \langle z_1, z_2\rangle : T', \Delta_2 \vdash M : S \rightsquigarrow e$ *and* $\Delta_1 \vdash V_2 : T_3 \rightsquigarrow v$*, where* M *and* Δ_2 *are* $z.L_1$*-only-projection, then* $\Delta_1, \Delta_2[z.L_1 \mapsto V_2] \vdash M[z.L_1 \mapsto V_2] : S[z.L_1 \mapsto V_2] \rightsquigarrow e[z_1 \mapsto v]$.

(4) If $\Delta_1, z = \langle z_1, z_2\rangle : T', \Delta_2 \vdash S \rightsquigarrow e$ *and* $\Delta_1 \vdash V_2 : T_1 \rightsquigarrow v$*, where* S *and* Δ_2 *are* $z.L_1$*-only-projection, then* $\Delta_1, \Delta_2[z.L_1 \mapsto V_2] \vdash S[z.L_1 \mapsto V_2] \rightsquigarrow e[z_1 \mapsto v]$.

Proof. By mutual induction on the derivation of translation. The cases for (2,3,4) are trivial. We only show proof cases for (1). For source meta-variables, we use E^* to denote $E[z.L_1 \mapsto V_2]$, Δ^* to denote $\Delta[z.L_1 \mapsto V_2]$, etc. For target meta-variables, we use e^* to denote $e[z_1 \mapsto v]$.

- Case TR-Ax: Noting that $\mathsf{Type}^* = \mathsf{Type}$, trivial by IH.
- Case TR-Var, TR-VarM: If $x = z$, then x is not $z.L_1$-only-projection. Thus $x^* = x$, the conclusion is trivial by IH.
- Case TR-Pi:

${T_1}^*$ is not a trait	by definition since V cannot be a trait (see Figure 6.4, similarly for other cases)
$T_1, T_2, (\Delta_2, x : T_2)$ are $z.L_1$-only-projection	by definition
$\Delta_1, \Delta_2^* \vdash {T_1}^* : \mathsf{Type} \rightsquigarrow {A_1}^*$	by IH
$\Delta_1, \Delta_2^*, x : {T_1}^* \vdash {T_2}^* : \mathsf{Type} \rightsquigarrow {A_2}^*$	by IH
$\Delta_1, \Delta_2^* \vdash (x : {T_1}^*) \to {T_2}^* : \mathsf{Type} \rightsquigarrow \Pi x : {A_1}^*.\ {A_2}^*$	by TR-Pi
$\Delta_1, \Delta_2^* \vdash ((x : T_1) \to T_2)^* : \mathsf{Type} \rightsquigarrow (\Pi x : A_1.\ A_2)^*$	by definition

- Case TR-PiM:

${T_1}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$T_1, T_2, (\Delta_2, x = \langle x_1, x_2\rangle : T_1)$ are $z.L_1$-only-projection	by definition
$\Delta_1, \Delta_2^* \vdash {T_1}^* : \mathsf{Type} \rightsquigarrow (\Sigma x_1 : A.\ B)^*$	by IH
$\Delta_1, \Delta_2^*, x = \langle x_1, x_2\rangle : {T_1}^* \vdash {T_2}^* : \mathsf{Type} \rightsquigarrow C^*$	by IH
$\Delta_1, \Delta_2^* \vdash (x : {T_1}^*) \to {T_2}^* : \mathsf{Type} \rightsquigarrow \Pi y : (\Sigma x_1 : A.\ B)^*.\ (\lambda x_1 : A^*.\ C^*)\,(y.1)$	by TR-PiM
$\Delta_1, \Delta_2^* \vdash ((x : T_1) \to T_2)^* : \mathsf{Type} \rightsquigarrow (\Pi y : (\Sigma x_1 : A.\ B).\ (\lambda x_1 : A.\ C)\,(y.1))^*$	by definition

- Case TR-App:

${T_1}^*$ is not a trait	by definition
E_1, E_2 are $z.L_1$-only-projection	by definition
$\Delta_1, \Delta_2^* \vdash {E_1}^* : {T_1}^* \to {T_2}^* \rightsquigarrow {e_1}^*$	by IH
$\Delta_1, \Delta_2^* \vdash {E_2}^* : {T_1}^* \rightsquigarrow {e_2}^*$	by IH
$\Delta_1, \Delta_2^* \vdash {E_1}^*\,{E_2}^* : {T_2}^* \rightsquigarrow {e_1}^*\,{e_2}^*$	by TR-App
$\Delta_1, \Delta_2^* \vdash (E_1\ E_2)^* : {T_2}^* \rightsquigarrow (e_1\ e_2)^*$	by definition

- Case TR-AppV:

${T_1}^*$ is not a trait	by definition
E, V are $z.L_1$-only-projection	by definition
V^* is a value	by value substitution
$\Delta_1, \Delta_2^* \vdash E^* : (x : {T_1}^*) \to {T_2}^* \rightsquigarrow {e_1}^*$	by IH
$\Delta_1, \Delta_2^* \vdash V^* : {T_1}^* \rightsquigarrow {e_2}^*$	by IH
$\Delta_1, \Delta_2^* \vdash E^*\,V^* : {T_2}^*[x \mapsto V^*] \rightsquigarrow {e_1}^*\,{e_2}^*$	by TR-AppV
$\Delta_1, \Delta_2^* \vdash (E\ V)^* : (T_2[x \mapsto V])^* \rightsquigarrow (e_1\ e_2)^*$	by definition

- Case TR-AppX: $z = y$ is impossible similarly to case TR-VarM. When $z \neq y$, $y^* = y$.

$T_1{}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
E is $z.L_1$-only-projection	by definition
$\Delta_1, \Delta_2^* \vdash E^* : (x : T_1{}^*) \to T_2{}^* \rightsquigarrow e^*$	by IH
$\Delta_1, \Delta_2^* \vdash y : T_1{}^* \rightsquigarrow \langle y_1, y_2 \;\mathsf{as}\; \Sigma x_1 : A.\ B\rangle^*$	by IH
$\Delta_1, \Delta_2^* \vdash E^*\ y : T_2{}^*[x \mapsto y] \rightsquigarrow \mathsf{cast}^2_\downarrow\ (e^*\ \langle y_1, y_2 \;\mathsf{as}\; \Sigma x_1 : A.\ B\rangle^*)$	by TR-AppX
$\Delta_1, \Delta_2^* \vdash (E\ y)^* : (T_2[x \mapsto y])^* \rightsquigarrow (\mathsf{cast}^2_\downarrow\ (e\ \langle y_1, y_2 \;\mathsf{as}\; \Sigma x_1 : A.\ B\rangle))^*$	by definition

- Case TR-AppM:

$T_1{}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$V_1{}^* = \mathbf{obj}\,\{\,\mathsf{type}\,L = V^*; M^*\}\ \mathbf{as}\ T_1{}^*$	by definition
E, V_1 are $z.L_1$-only-projection	by definition
$V_1{}^*$ is a value	by value subst.
$\Delta_1, \Delta_2^* \vdash E^* : (x : T_1{}^*) \to T_2{}^* \rightsquigarrow e^*$	by IH
$\Delta_1, \Delta_2^* \vdash V_1{}^* : T_1{}^* \rightsquigarrow \langle e_1, e_2 \;\mathsf{as}\; \Sigma x_1 : A.\ B\rangle^*$	by IH
$\Delta_1, \Delta_2^* \vdash E^*\ V_1{}^* : T_2{}^*[x.L \mapsto V^*] \rightsquigarrow \mathsf{cast}^2_\downarrow\ (e^*\ \langle e_1, e_2 \;\mathsf{as}\; \Sigma x_1 : A.\ B\rangle^*)$	by TR-AppM
$\Delta_1, \Delta_2^* \vdash (E\ V_1)^* : (T_2[x.L \mapsto V])^* \rightsquigarrow (\mathsf{cast}^2_\downarrow\ (e\ \langle e_1, e_2 \;\mathsf{as}\; \Sigma x_1 : A.\ B\rangle))^*$	by definition

- Case TR-Lam:

$T_1{}^*$ is not a trait	by definition
$T_1, E, (\Delta_2, x : T_1)$ are $z.L_1$-only-projection	by definition
$\Delta_1, \Delta_2^* \vdash T_1{}^* : \mathsf{Type} \rightsquigarrow A^*$	by IH
$\Delta_1, \Delta_2^*, x : T_1{}^* \vdash E^* : T_2{}^* \rightsquigarrow e^*$	by IH
$\Delta_1, \Delta_2^* \vdash \lambda(x : T_1{}^*) \Rightarrow E^* : (x : T_1{}^*) \to T_2{}^* \rightsquigarrow \lambda x : A^*.\ e^*$	by TR-Lam
$\Delta_1, \Delta_2^* \vdash (\lambda(x : T_1) \Rightarrow E)^* : ((x : T_1) \to T_2)^* \rightsquigarrow (\lambda x : A.\ e)^*$	by definition

- Case TR-LamM:

$T_1{}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$T_1, E, (\Delta_2, x : T_1)$ are $z.L_1$-only-projection	by definition
$\Delta_1, \Delta_2^* \vdash T_1{}^* : \mathsf{Type} \rightsquigarrow (\Sigma x_1 : A.\ B)^*$	by IH
$\Delta_1, \Delta_2^*, x = \langle x_1, x_2\rangle : T_1{}^* \vdash E^* : T_2{}^* \rightsquigarrow e^*$	by IH
$\Delta_1, \Delta_2^* \vdash \lambda(x : T_1{}^*) \Rightarrow E^* : (x : T_1{}^*) \to T_2{}^* \rightsquigarrow e_1$	by TR-LamM
where $e_1 = \lambda y : (\Sigma x_1 : A.\ B)^*.\ \mathbf{open}\ y\ \mathbf{as}\ \langle x_1, x_2\rangle\ \mathbf{in}\ e^*$	
$\Delta_1, \Delta_2^* \vdash (\lambda(x : T_1) \Rightarrow E)^* : ((x : T_1) \to T_2)^* \rightsquigarrow {e_2}^*$	by definition
where $e_2 = \lambda y : (\Sigma x_1 : A.\ B).\ \mathbf{open}\ y\ \mathbf{as}\ \langle x_1, x_2\rangle\ \mathbf{in}\ e$	

- Case TR-Mod:

T^* is not a trait	by definition
$T_1{}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
V, M, T_1 are $z.L_1$-only-projection	by definition
$M^*[L \mapsto V^*]$ is $z.L_1$-only-projection	by definition
$\Delta_1, \Delta_2^* \vdash V^* : T^* \rightsquigarrow {e_1}^*$	by IH
$\Delta_1, \Delta_2^* \vdash M^*[L \mapsto V^*] : S^*[L \mapsto V^*] \rightsquigarrow {e_2}^*$	by IH
$\Delta_1, \Delta_2^* \vdash T_1{}^* : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A^*.\ B^*$	by IH
$\Delta_1, \Delta_2^* \vdash \mathbf{obj}\,\{\,\mathsf{type}\,L = V^*; M^*\}\ \mathbf{as}\ T_1{}^* : T_1{}^* \rightsquigarrow \langle {e_1}^*, {e_2}^* \;\mathsf{as}\; \Sigma x_1 : A^*.\ B^*\rangle$	by TR-Mod
$\Delta_1, \Delta_2^* \vdash (\mathbf{obj}\,\{\,\mathsf{type}\,L = V; M\}\ \mathbf{as}\ T_1)^* : T_1{}^* \rightsquigarrow \langle e_1, e_2 \;\mathsf{as}\; \Sigma x_1 : A.\ B\rangle^*$	by definition

- Case TR-Sig:

T^* is not a trait	by definition
$T, S, (\Delta_2, L : T)$ are $z.L_1$-only-projection	by definition
$\Delta_1, \Delta_2^* \vdash T^* : \mathsf{Type} \rightsquigarrow A^*$	by IH
$\Delta_1, \Delta_2^*, L : T^* \vdash S^* \rightsquigarrow B^*$	by IH
$\Delta_1, \Delta_2^* \vdash \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\} : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A^*.\ B^*[L \mapsto x_1]$	by TR-Sig
$\Delta_1, \Delta_2^* \vdash (\mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\})^* : \mathsf{Type} \rightsquigarrow (\Sigma x_1 : A.\ B[L \mapsto x_1])^*$	by definition

- Case TR-ProjT:

- Case $x = z$: we have $L = L_1$, otherwise $x.L$ is not $z.L_1$-only-projection. Thus, $x.L = z.L_1$ and $(x.L)^* = V_2$.

$\Delta_1, z = \langle z_1, z_2 \rangle : T', \Delta_2 \vdash z.L_1 : T_3 \rightsquigarrow z_1$	by TR-ProjT
$\Delta_1, z = \langle z_1, z_2 \rangle : T', \Delta_2 \vdash x.L : T \rightsquigarrow x_1$	by condition
$T_3 = T, z_1 = x_1$	by uniqueness
${x_1}^* = v$	by equality
$T_3 = {T_3}^*$	by $z \notin \mathsf{FV}(T_3)$
$\Delta_1 \vdash V_2 : {T_3}^* \rightsquigarrow v$	by equality
$\Delta_1, \Delta_2^* \vdash V_2 : {T_3}^* \rightsquigarrow v$	by weakening
$\Delta_1, \Delta_2^* \vdash (x.L)^* : T^* \rightsquigarrow {x_1}^*$	by equality

- Case $x \neq z$: we have $(x.L)^* = x.L$.

${T_1}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$\Delta_1, \Delta_2^* \vdash x : {T_1}^* \rightsquigarrow \langle x_1, x_2 \;\mathsf{as}\; \Sigma x_1 : A^*.\ B^* \rangle$	by IH
$\Delta_1, \Delta_2^* \vdash x.L : T^* \rightsquigarrow x_1$	by TR-ProjT
$\Delta_1, \Delta_2^* \vdash (x.L)^* : T^* \rightsquigarrow {x_1}^*$	by $x^* = x$

- Case TR-Proj: Note that $x \neq z$ otherwise $x.l$ is not $z.L_1$-only-projection. Thus $(x.l)^* = x.l$.

${T_1}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$\Delta_1, \Delta_2^* \vdash x : {T_1}^* \rightsquigarrow \langle x_1, x_2 \;\mathsf{as}\; \Sigma x_1 : A^*.\ B^* \rangle$	by IH
$\mathsf{val}\, l : {T_2}^* \in S^*$	by definition
$\Delta_1, \Delta_2^* \vdash x.l : {T_2}^*[L \mapsto x.L] \rightsquigarrow x_2.l$	by TR-Proj
$\Delta_1, \Delta_2^* \vdash (x.l)^* : ({T_2}[L \mapsto x.L])^* \rightsquigarrow (x_2.l)^*$	by $x^* = x$

□

Definition A.3.2 (Non-projection Terms)**.** *We call E x-*non-projection *if it does not contain any projection form of x (i.e. $x.L$ or $x.l$ where x occurs free in E). Similarly, bindings Δ, M and S are x-non-projection if they only contain x-non-projection terms.*

Lemma A.3.7 (Value Substitution)**.** *If V_1 is a x-non-projection value and V_2 is a value, then $V_1[x \mapsto V_2]$ is still a value.*

Proof. Trivial by induction on the form of V_1. □

Lemma A.3.8 (Typing Substitution)**.**

(1) If $\Delta_1, z : T_1, \Delta_2 \vdash E : T \rightsquigarrow e$ and $\Delta_1 \vdash V_2 : T_1 \rightsquigarrow v$, where E and Δ_2 are z-non-projection, then $\Delta_1, \Delta_2[z \mapsto V_2] \vdash E[z \mapsto V_2] : T[z \mapsto V_2] \rightsquigarrow e[z \mapsto v]$.

(2) If $\vdash \Delta_1, z : T_1, \Delta_2 \rightsquigarrow \Gamma_1, z : A, \Gamma_2$ and $\Delta_1 \vdash V_2 : T_1 \rightsquigarrow v$, where Δ_2 is z-non-projection, then $\vdash \Delta_1, \Delta_2[z \mapsto V_2] \rightsquigarrow \Gamma_1, \Gamma_2[z \mapsto v]$.

(3) If $\Delta_1, z : T_1, \Delta_2 \vdash M : S \rightsquigarrow e$ and $\Delta_1 \vdash V_2 : T_1 \rightsquigarrow v$, where M and Δ_2 are z-non-projection, then $\Delta_1, \Delta_2[z \mapsto V_2] \vdash M[z \mapsto V_2] : S[z \mapsto V_2] \rightsquigarrow e[z \mapsto v]$.

(4) If $\Delta_1, z : T_1, \Delta_2 \vdash S \rightsquigarrow e$ and $\Delta_1 \vdash V_2 : T_1 \rightsquigarrow v$, where S and Δ_2 are z-non-projection, then $\Delta_1, \Delta_2[z \mapsto V_2] \vdash S[z \mapsto V_2] \rightsquigarrow e[z \mapsto v]$.

Proof. By mutual induction on the derivation of translation. The cases for (2,3,4) are trivial. We only show proof cases for (1). For source meta-variables, we use E^* to denote $E[z \mapsto V_2]$, Δ^* to denote $\Delta[z \mapsto V_2]$, etc. For target meta-variables, we use e^* to denote $e[z \mapsto v]$.

- Case TR-Ax: Trivial by IH.

- Case TR-Var: Suppose $E = x$,
  - Case $z = x$: Immediate by $x \notin \mathsf{FV}(T)$ and weakening.
  - Case $z \neq x$: We have $x^* = x$ and $x : T \in \Delta_1, z : T_1, \Delta_2$. If $x : T \in \Delta_1$, then $z \notin \mathsf{dom}(\Delta_1)$ and $z \notin \mathsf{FV}(T)$. Thus, $T^* = T$. The conclusion holds immediately by weakening. Otherwise, $x : T \in \Delta_2$ and $x : T^* \in \Delta_2^*$. Note that Δ_2 is z-non-projection, then $\vdash \Delta_1, \Delta_2^* \rightsquigarrow \Gamma$. Thus, the conclusion holds by TR-Var.
- Case TR-VarM: Suppose $E = y$,
  - Case $z = x$: Impossible. $x : T_1 \in \Delta$ and $x = \langle x_1, x_2 \rangle : T_1 \in \Delta$ are mismatched.
  - Case $z \neq x$: We have $x^* = y$ and $x = \langle x_1, x_2 \rangle : T \in \Delta_1, z : T_1, \Delta_2$. If $x = \langle x_1, x_2 \rangle : T \in \Delta_1$, then $z \notin \mathsf{FV}(\Delta_1)$ and $z \notin \mathsf{FV}(T)$. Thus, $T^* = T$. The conclusion holds immediately by weakening. Otherwise, $x = \langle x_1, x_2 \rangle : T \in \Delta_2$ and $x = \langle x_1, x_2 \rangle : T^* \in \Delta_2^*$. By IH, we have $\vdash \Delta_1, \Delta_2^* \rightsquigarrow \Gamma_1, \Gamma_2^*$. By inversion of TRW-Mod, we have $\Delta_1, \Delta_2^* \vdash T^* : \mathsf{Type} \rightsquigarrow (\Sigma x_1 : A.\ B)^*$ The conclusion holds by TR-VarM.
- Case TR-Pi:

${T_1}^*$ is not a trait	by definition
$T_1, T_2, (\Delta_2, x : T_2)$ are z-non-projection	by definition
$\Delta_1, \Delta_2^* \vdash {T_1}^* : \mathsf{Type} \rightsquigarrow {A_1}^*$	by IH
$\Delta_1, \Delta_2^*, x : {T_1}^* \vdash {T_2}^* : \mathsf{Type} \rightsquigarrow {A_2}^*$	by IH
$\Delta_1, \Delta_2^* \vdash (x : {T_1}^*) \to {T_2}^* : \mathsf{Type} \rightsquigarrow \Pi x : {A_1}^*.\ {A_2}^*$	by TR-Pi
$\Delta_1, \Delta_2^* \vdash ((x : T_1) \to T_2)^* : \mathsf{Type} \rightsquigarrow (\Pi x : A_1.\ A_2)^*$	by definition

- Case TR-PiM:

${T_1}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$T_1, T_2, (\Delta_2, x = \langle x_1, x_2 \rangle : T_1)$ are z-non-projection	by definition
$\Delta_1, \Delta_2^* \vdash {T_1}^* : \mathsf{Type} \rightsquigarrow (\Sigma x_1 : A.\ B)^*$	by IH
$\Delta_1, \Delta_2^*, x = \langle x_1, x_2 \rangle : {T_1}^* \vdash {T_2}^* : \mathsf{Type} \rightsquigarrow C^*$	by IH
$\Delta_1, \Delta_2^* \vdash (x : {T_1}^*) \to {T_2}^* : \mathsf{Type} \rightsquigarrow \Pi y : (\Sigma x_1 : A.\ B)^*.\ (\lambda x_1 : A^*.\ C^*)\,(y.1)$	by TR-PiM
$\Delta_1, \Delta_2^* \vdash ((x : T_1) \to T_2)^* : \mathsf{Type} \rightsquigarrow (\Pi y : (\Sigma x_1 : A.\ B).\ (\lambda x_1 : A.\ C)\,(y.1))^*$	by definition

- Case TR-App:

${T_1}^*$ is not a trait	by definition
E_1, E_2 are z-non-projection	by definition
$\Delta_1, \Delta_2^* \vdash {E_1}^* : {T_1}^* \to {T_2}^* \rightsquigarrow {e_1}^*$	by IH
$\Delta_1, \Delta_2^* \vdash {E_2}^* : {T_1}^* \rightsquigarrow {e_2}^*$	by IH
$\Delta_1, \Delta_2^* \vdash {E_1}^*\,{E_2}^* : {T_2}^* \rightsquigarrow {e_1}^*\,{e_2}^*$	by TR-App
$\Delta_1, \Delta_2^* \vdash (E_1\,E_2)^* : {T_2}^* \rightsquigarrow (e_1\,e_2)^*$	by definition

- Case TR-AppV:

${T_1}^*$ is not a trait	by definition
E, V are z-non-projection	by definition
V^* is a value	by value substitution
$\Delta_1, \Delta_2^* \vdash E^* : (x : {T_1}^*) \to {T_2}^* \rightsquigarrow {e_1}^*$	by IH
$\Delta_1, \Delta_2^* \vdash V^* : {T_1}^* \rightsquigarrow {e_2}^*$	by IH
$\Delta_1, \Delta_2^* \vdash E^*\,V^* : {T_2}^*[x \mapsto V^*] \rightsquigarrow {e_1}^*\,{e_2}^*$	by TR-AppV
$\Delta_1, \Delta_2^* \vdash (E\,V)^* : (T_2[x \mapsto V])^* \rightsquigarrow (e_1\,e_2)^*$	by definition

- Case TR-AppX: $z = y$ is impossible similarly to case TR-VarM. When $z \neq y$, $y^* = y$.

${T_1}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
E is z-non-projection	by definition
$\Delta_1, \Delta_2^* \vdash E^* : (x : {T_1}^*) \to {T_2}^* \rightsquigarrow e^*$	by IH
$\Delta_1, \Delta_2^* \vdash y : {T_1}^* \rightsquigarrow \langle y_1, y_2\ \mathsf{as}\ \Sigma x_1 : A.\ B \rangle^*$	by IH
$\Delta_1, \Delta_2^* \vdash E^*\,y : {T_2}^*[x \mapsto y] \rightsquigarrow \mathsf{cast}^2_{\downarrow}\,(e^*\,\langle y_1, y_2\ \mathsf{as}\ \Sigma x_1 : A.\ B \rangle^*)$	by TR-AppX
$\Delta_1, \Delta_2^* \vdash (E\,y)^* : (T_2[x \mapsto y])^* \rightsquigarrow (\mathsf{cast}^2_{\downarrow}\,(e\,\langle y_1, y_2\ \mathsf{as}\ \Sigma x_1 : A.\ B \rangle))^*$	by definition

- Case TR-AppM:

$T_1{}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$V_1{}^* = \mathbf{obj}\,\{\,\mathsf{type}\,L = V^*; M^*\}\,\mathbf{as}\,T_1{}^*$	by definition
E, V_1 are z-non-projection	by definition
$V_1{}^*$ is a value	by value subst.
$\Delta_1, \Delta_2^* \vdash E^* : (x : T_1{}^*) \to T_2{}^* \rightsquigarrow e^*$	by IH
$\Delta_1, \Delta_2^* \vdash V_1{}^* : T_1{}^* \rightsquigarrow \langle e_1, e_2\ \mathsf{as}\ \Sigma x_1 : A.\ B\rangle^*$	by IH
$\Delta_1, \Delta_2^* \vdash E^*\ V_1{}^* : T_2{}^*[x.L \mapsto V^*] \rightsquigarrow \mathsf{cast}^2_\downarrow\ (e^*\ \langle e_1, e_2\ \mathsf{as}\ \Sigma x_1 : A.\ B\rangle^*)$	by TR-AppM
$\Delta_1, \Delta_2^* \vdash (E\ V_1)^* : (T_2[x.L \mapsto V])^* \rightsquigarrow (\mathsf{cast}^2_\downarrow\ (e\ \langle e_1, e_2\ \mathsf{as}\ \Sigma x_1 : A.\ B\rangle))^*$	by definition

- Case TR-Lam:

$T_1{}^*$ is not a trait	by definition
$T_1, E, (\Delta_2, x : T_1)$ are z-non-projection	by definition
$\Delta_1, \Delta_2^* \vdash T_1{}^* : \mathsf{Type} \rightsquigarrow A^*$	by IH
$\Delta_1, \Delta_2^*, x : T_1{}^* \vdash E^* : T_2{}^* \rightsquigarrow e^*$	by IH
$\Delta_1, \Delta_2^* \vdash \lambda(x : T_1{}^*) \Rightarrow E^* : (x : T_1{}^*) \to T_2{}^* \rightsquigarrow \lambda x : A^*.\ e^*$	by TR-Lam
$\Delta_1, \Delta_2^* \vdash (\lambda(x : T_1) \Rightarrow E)^* : ((x : T_1) \to T_2)^* \rightsquigarrow (\lambda x : A.\ e)^*$	by definition

- Case TR-LamM:

$T_1{}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$T_1, E, (\Delta_2, x : T_1)$ are z-non-projection	by definition
$\Delta_1, \Delta_2^* \vdash T_1{}^* : \mathsf{Type} \rightsquigarrow (\Sigma x_1 : A.\ B)^*$	by IH
$\Delta_1, \Delta_2^*, x = \langle x_1, x_2\rangle : T_1{}^* \vdash E^* : T_2{}^* \rightsquigarrow e^*$	by IH
$\Delta_1, \Delta_2^* \vdash \lambda(x : T_1{}^*) \Rightarrow E^* : (x : T_1{}^*) \to T_2{}^* \rightsquigarrow e_1$	by TR-LamM
where $e_1 = \lambda y : (\Sigma x_1 : A.\ B)^*.\ \mathbf{open}\ y\ \mathbf{as}\ \langle x_1, x_2\rangle\ \mathbf{in}\ e^*$	
$\Delta_1, \Delta_2^* \vdash (\lambda(x : T_1) \Rightarrow E)^* : ((x : T_1) \to T_2)^* \rightsquigarrow e_2{}^*$	by definition
where $e_2 = \lambda y : (\Sigma x_1 : A.\ B).\ \mathbf{open}\ y\ \mathbf{as}\ \langle x_1, x_2\rangle\ \mathbf{in}\ e$	

- Case TR-Mod:

T^* is not a trait	by definition
$T_1{}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
V, M, T_1 are z-non-projection	by definition
$M^*[L \mapsto V^*]$ is z-non-projection	by definition
$\Delta_1, \Delta_2^* \vdash V^* : T^* \rightsquigarrow e_1{}^*$	by IH
$\Delta_1, \Delta_2^* \vdash M^*[L \mapsto V^*] : S^*[L \mapsto V^*] \rightsquigarrow e_2{}^*$	by IH
$\Delta_1, \Delta_2^* \vdash T_1{}^* : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A^*.\ B^*$	by IH
$\Delta_1, \Delta_2^* \vdash \mathbf{obj}\,\{\,\mathsf{type}\,L = V^*; M^*\}\,\mathbf{as}\,T_1{}^* : T_1{}^* \rightsquigarrow \langle e_1{}^*, e_2{}^*\ \mathsf{as}\ \Sigma x_1 : A^*.\ B^*\rangle$	by TR-Mod
$\Delta_1, \Delta_2^* \vdash (\mathbf{obj}\,\{\,\mathsf{type}\,L = V; M\}\,\mathbf{as}\,T_1)^* : T_1{}^* \rightsquigarrow \langle e_1, e_2\ \mathsf{as}\ \Sigma x_1 : A.\ B\rangle^*$	by definition

- Case TR-Sig:

T^* is not a trait	by definition
$T, S, (\Delta_2, L : T)$ are z-non-projection	by definition
$\Delta_1, \Delta_2^* \vdash T^* : \mathsf{Type} \rightsquigarrow A^*$	by IH
$\Delta_1, \Delta_2^*, L : T^* \vdash S^* \rightsquigarrow B^*$	by IH
$\Delta_1, \Delta_2^* \vdash \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\} : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A^*.\ B^*[L \mapsto x_1]$	by TR-Sig
$\Delta_1, \Delta_2^* \vdash (\mathbf{trait}\,\{\,\mathsf{type}\,L : T; S\})^* : \mathsf{Type} \rightsquigarrow (\Sigma x_1 : A.\ B[L \mapsto x_1])^*$	by definition

- Case TR-ProjT: Note that $x \neq z$ otherwise $x.L$ is not x-non-projection. Thus $x^* = x$.

$T_1{}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$\Delta_1, \Delta_2^* \vdash x : T_1{}^* \rightsquigarrow \langle x_1, x_2\ \mathsf{as}\ \Sigma x_1 : A^*.\ B^*\rangle$	by IH
$\Delta_1, \Delta_2^* \vdash x.L : T^* \rightsquigarrow x_1$	by TR-ProjT
$\Delta_1, \Delta_2^* \vdash (x.L)^* : T^* \rightsquigarrow x_1{}^*$	by $x^* = x$

- Case TR-Proj: Note that $x \neq z$ otherwise $x.l$ is not x-non-projection. Thus $x^* = x$.

$T_1{}^* = \mathbf{trait}\,\{\,\mathsf{type}\,L : T^*; S^*\}$	by definition
$\Delta_1, \Delta_2^* \vdash x : T_1{}^* \rightsquigarrow \langle x_1, x_2\ \mathsf{as}\ \Sigma x_1 : A^*.\ B^*\rangle$	by IH
$\mathsf{val}\, l : T_2{}^* \in S^*$	by definition
$\Delta_1, \Delta_2^* \vdash x.l : T_2{}^*[L \mapsto x.L] \rightsquigarrow x_2.l$	by TR-Proj
$\Delta_1, \Delta_2^* \vdash (x.l)^* : (T_2[L \mapsto x.L])^* \rightsquigarrow (x_2.l)^*$	by $x^* = x$

□

Lemma A.3.9 (Correctness of Types). *If* $\Delta \vdash E : T \rightsquigarrow e$, *then exists* A *such that* $\Delta \vdash T : \mathsf{Type} \rightsquigarrow A$.

Proof. By induction on the derivation of $\Delta \vdash E : T \rightsquigarrow e$. Most cases are trivial by induction hypothesis. We only show interesting cases where the conclusion type T involves substitution.

- Case TR-AppV:

T_2 is x-non-projection	by T_1 is not a trait
$\Delta, x : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow A_2$	by inversion of TR-Pi
$\Delta \vdash V : T_1 \rightsquigarrow e_2$	by premise
e_2 is a value	by value translation
$\Delta \vdash T_2[x \mapsto V] : \mathsf{Type} \rightsquigarrow A_2[x \mapsto e_2]$	by substitution

- Case TR-AppX:

$\Delta, x = \langle x_1, x_2\rangle : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow C$	by inversion of TR-PiM
T_2 is $x.L$-only-projection	by inversion of TR-PiM
$\Delta \vdash y.L : T \rightsquigarrow y_1$	by TR-ProjT
$\Delta \vdash T_2[x.L \mapsto y.L] : \mathsf{Type} \rightsquigarrow C[x_1 \mapsto y_1]$	by projection substitution
$\Delta \vdash T_2[x \mapsto y] : \mathsf{Type} \rightsquigarrow C[x_1 \mapsto y_1]$	

- Case TR-AppM:

$\Delta, x = \langle x_1, x_2\rangle : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow C$	by inversion of TR-PiM
T_2 is $x.L$-only-projection	by inversion of TR-PiM
$\Delta \vdash V : T \rightsquigarrow e_1$	by inversion of TR-Mod
e_1 is a value	by translation of values
$\Delta \vdash T_2[x.L \mapsto V] : \mathsf{Type} \rightsquigarrow C[x_1 \mapsto e_1]$	by projection substitution

- Case TR-Proj:

$\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B$	by IH
T is not a trait	by inversion of TR-Sig
$\Delta, L : T \vdash T_2 : \mathsf{Type} \rightsquigarrow A_2$	by inversion of TRS-Sig
T_2 is L-non-projection	by T is not a trait
$\Delta \vdash x.L : T \rightsquigarrow x_1$	by TR-ProjT
$\Delta \vdash T_2[L \mapsto x.L] : \mathsf{Type} \rightsquigarrow A_2[L \mapsto x_1]$	by substitution

□

Lemma A.3.10 (Soundness of Translation).

(1) If $\vdash \Delta \rightsquigarrow \Gamma$, *then* $\vdash \Gamma$.

(2) If $\vdash \Delta \rightsquigarrow \Gamma$ *and* $\Delta \vdash S \rightsquigarrow A$, *then* $\Gamma \vdash A : \star$.

(3) Given $\vdash \Delta \rightsquigarrow \Gamma$, *if* $\Delta \vdash E : T \rightsquigarrow e$ *and* $\Delta \vdash T : \mathsf{Type} \rightsquigarrow A$, *then* $\Gamma \vdash e : A$.

Proof. By simultaneous induction on the derivation of $\Delta \vdash E : T \rightsquigarrow e$, $\vdash \Delta \rightsquigarrow \Gamma$ and $\Delta \vdash M : S \rightsquigarrow e$. The cases in (1) and (2) are trivial. We only show the proofs regarding (3).

- Case TR-Ax:

$\vdash \Gamma$	by IH
$\Gamma \vdash \star : \star$	by SP-Ax

- Case TR-Var:

$x : A \in \Gamma$	by TRW-Cons
$\Delta \vdash T : \mathsf{Type} \rightsquigarrow A$	by TRW-Cons
$\vdash \Gamma$	by IH
$\Gamma \vdash x : A$	by SP-Var

- Case TR-VarM:

$x_1 : A, x_2 : B \in \Gamma$	by TRW-Mod
$\Gamma \vdash x_1 : A$	by SP-Var
$\Gamma \vdash x_2 : B$	by SP-Var
$\Gamma, x_1 : A \vdash B : \star$	by well-formedness of $\Sigma x_1 : A.\ B$
$\Gamma \vdash \langle x_1, x_2 \text{ as } \Sigma x_1 : A.\ B\rangle : \Sigma x_1 : A.\ B$	by SP-Pair

- Case TR-Pi:

$\Gamma \vdash A_1 : \star$	by IH
$\Gamma, x : A_1 \vdash A_2 : \star$	by IH and TRW-Cons
$\Gamma \vdash \Pi x : A_1.\ A_2 : \star$	by SP-Prod

- Case TR-PiM:

$\Gamma \vdash \Sigma x_1 : A.\ B : \star$	by IH
$\Gamma, x_1 : A, x_2 : B \vdash C : \star$	by IH and TRW-Mod
$x_2 \notin \mathsf{FV}(C)$	by T_2 is $x.L$-only-projection
$\Gamma \vdash \lambda x_1 : A.\ C : A \to \star$	by IH and SP-Abs
$\Gamma, y : \Sigma x_1 : A.\ B \vdash y.1 : A$	by SP-Fst
$\Gamma, y : \Sigma x_1 : A.\ B \vdash (\lambda x_1 : A.\ C)\,(y.1) : A$	by SP-App
$\Gamma \vdash \Pi y : (\Sigma x_1 : A.\ B).\ (\lambda x_1 : A.\ C)\,(y.1) : \star$	by SP-Prod

- Case TR-App:

$\Delta \vdash T_1 \to T_2 : \mathsf{Type} \rightsquigarrow A_1 \to A_2$	by correctness of types
$\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow A_1$	by inversion of TR-Pi
$\Delta \vdash T_2 : \mathsf{Type} \rightsquigarrow A_2$	by inversion of TR-Pi
$\Gamma \vdash e_1 : A_1 \to A_2$	by IH
$\Gamma \vdash e_2 : A_1$	by IH
$\Gamma \vdash e_1\ e_2 : A_2$	by SP-App

- Case TR-AppV:

$\Delta \vdash (x : T_1) \to T_2 : \mathsf{Type} \rightsquigarrow \Pi x : A_1.\ A_2$	by correctness of types
$\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow A_1$	by inversion of TR-Pi
$\Delta, x : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow A_2$	by inversion of TR-Pi
$\Gamma \vdash e_1 : \Pi x : A_1.\ A_2$	by IH
$\Gamma \vdash e_2 : A_1$	by IH
e_2 is a value	by translation of values
$\Gamma \vdash e_1\ e_2 : A_2[x \mapsto e_2]$	by SP-AppV

- Case TR-AppX:

$\Delta \vdash (x : T_1) \to T_2 : \mathsf{Type} \rightsquigarrow A_1$	by correctness of types
where $A_1 = \Pi y : (\Sigma x_1 : A.\ B).\ (\lambda x_1 : A.\ C)\ (y.1)$	
$\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B$	by inversion of TR-PiM
T_2 is $x.L$-only-projection	by inversion of TR-PiM
$y = \langle y_1, y_2 \rangle : T_1 \in \Delta$	by inversion of TR-VarM
$y_1 : A, y_2 : B \in \Gamma$	by inversion of TRW-Mod
$\Delta \vdash y.L : T \rightsquigarrow y_1$	by TR-ProjT
$\Delta, x = \langle x_1, x_2 \rangle : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow C$	by inversion of TR-PiM
$\Delta \vdash T_2[x.L \mapsto y.L] : \mathsf{Type} \rightsquigarrow C[x_1 \mapsto y_1]$	by projection substitution
$\Delta \vdash T_2[x \mapsto y] : \mathsf{Type} \rightsquigarrow C[x_1 \mapsto y_1]$	
$\Gamma \vdash e : \Pi y : (\Sigma x_1 : A.\ B).\ (\lambda x_1 : A.\ C)\ (y.1)$	by IH
$\Gamma \vdash \langle y_1, y_2 \ \mathsf{as}\ \Sigma x_1 : A.\ B \rangle : \Sigma x_1 : A.\ B$	by IH
$\Gamma \vdash \mathsf{cast}^2_{\downarrow}\ (e\ \langle y_1, y_2 \ \mathsf{as}\ \Sigma x_1 : A.\ B \rangle) : C[x_1 \mapsto y_1]$	by SP-CastDn and SP-AppV

- Case TR-AppM:

$\Delta \vdash (x : T_1) \to T_2 : \mathsf{Type} \rightsquigarrow A_1$	by correctness of types
where $A_1 = \Pi y : (\Sigma x_1 : A.\ B).\ (\lambda x_1 : A.\ C)\ (y.1)$	
$\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B$	by inversion of TR-PiM
$\Delta, x = \langle x_1, x_2 \rangle : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow C$	by inversion of TR-PiM
T_2 is $x.L$-only-projection	by inversion of TR-PiM
$\Delta \vdash V : T \rightsquigarrow e_1$	by inversion of TR-Mod
e_1 is a value	by translation of values
$\Delta \vdash T_2[x.L \mapsto V] : \mathsf{Type} \rightsquigarrow C[x_1 \mapsto e_1]$	by projection substitution
$\Gamma \vdash e : \Pi y : (\Sigma x_1 : A.\ B).\ (\lambda x_1 : A.\ C)\ (y.1)$	by IH
$\Gamma \vdash \langle e_1, e_2 \ \mathsf{as}\ \Sigma x_1 : A.\ B \rangle : \Sigma x_1 : A.\ B$	by IH
$\Gamma \vdash \mathsf{cast}^2_{\downarrow}\ (e\ \langle e_1, e_2 \ \mathsf{as}\ \Sigma x_1 : A.\ B \rangle) : C[x_1 \mapsto e_1]$	by SP-CastDn and SP-AppV

- Case TR-Lam:

$\Delta, x : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow B$	by correctness of types
$\Gamma \vdash A : \star$	by IH
$\Gamma, x : A \vdash e : B$	by IH
$\Delta \vdash (x : T_1) \to T_2 : \mathsf{Type} \rightsquigarrow \Pi x : A.\ B$	by TR-Pi
$\Gamma \vdash \lambda x : A.\ e : \Pi x : A.\ B$	by SP-Abs

- Case TR-LamM:

$\Delta, x = \langle x_1, x_2 \rangle : T_1 \vdash T_2 : \mathsf{Type} \rightsquigarrow C$	by correctness of types
$\Gamma \vdash \Sigma x_1 : A.\ B : \star$	by IH
$\Gamma, x_1 : A, x_2 : B \vdash e : C$	by IH
$\Delta \vdash (x : T_1) \to T_2 : \mathsf{Type} \rightsquigarrow A_1$	by TR-PiM
where $A_1 = \Pi y : (\Sigma x_1 : A.\ B).\ (\lambda x_1 : A.\ C)\ (y.1)$	
$x_2 \notin \mathsf{FV}(C)$	by T_2 is $x.L$-only-projection
$\Gamma, y : \Sigma x_1 : A.\ B \vdash \mathbf{open}\ y\ \mathbf{as}\ \langle x_1, x_2 \rangle\ \mathbf{in}\ e : (\lambda x_1 : A.\ C)\ (y.1)$	by SP-Open
$\Gamma \vdash \lambda y : (\Sigma x_1 : A.\ B).\ \mathbf{open}\ y\ \mathbf{as}\ \langle x_1, x_2 \rangle\ \mathbf{in}\ e : A_1$	by SP-Abs

- Case TR-Mod:

$\Delta \vdash T : \mathsf{Type} \rightsquigarrow A$	by inversion of TR-Sig
$\Delta, L : T \vdash S \rightsquigarrow B[x_1 \mapsto L]$	by inversion of TR-Sig
S is L-non-projection	by T is not a trait
$\Delta \vdash S[L \mapsto V] \rightsquigarrow B[x_1 \mapsto e_1]$	by substitution
$\Gamma \vdash e_1 : A$	by IH
$\Gamma \vdash e_2 : B[x_1 \mapsto e_1]$	by IH
e_1 is a value	by translation of values
$\Gamma \vdash \langle e_1, e_2 \ \mathsf{as}\ \Sigma x_1 : A.\ B \rangle : \Sigma x_1 : A.\ B$	by SP-Pair

- Case TR-Sig:

$\Gamma \vdash A : \star$	by IH
$\Gamma, L : A \vdash B : \star$	by IH
$\Gamma, x_1 : A \vdash B[L \mapsto x_1] : \star$	by substitution
$\Gamma \vdash \Sigma x_1 : A.\ B[L \mapsto x_1] : \star$	by SP-Sig

- Case TR-ProjT:

$x = \langle x_1, x_2 \rangle : T_1 \in \Delta$	by inversion of TR-VarM
$x_1 : A, x_2 : B \in \Gamma$	by inversion of TRW-Mod
$\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B$	by inversion of TRW-Mod
$\Delta \vdash T : \mathsf{Type} \rightsquigarrow A$	by inversion of TR-Sig
$\Gamma \vdash x_1 : A$	by SP-Var

- Case TR-Proj:

$x = \langle x_1, x_2 \rangle : T_1 \in \Delta$	by inversion of TR-VarM
$x_1 : A, x_2 : B \in \Gamma$	by inversion of TRW-Mod
$\Delta \vdash T_1 : \mathsf{Type} \rightsquigarrow \Sigma x_1 : A.\ B$	by inversion of TRW-Mod
$\Delta \vdash T : \mathsf{Type} \rightsquigarrow A$	by inversion of TR-Sig
$\Delta, L : T \vdash S \rightsquigarrow B$	by inversion of TR-Sig
$\Delta, L : T \vdash T_2 : \mathsf{Type} \rightsquigarrow A_2$	by inversion of TS-Sig
$l : A_2 \in B$	by inversion of TS-Sig
$\Gamma, L : A \vdash x_2.l : A_2$	by record projection
T_2 is L-non-projection	by T is not a trait
$\Delta \vdash x.L : T \rightsquigarrow x_1$	by TR-ProjT
$\Delta \vdash T_2[L \mapsto x.L] : \mathsf{Type} \rightsquigarrow A_2[L \mapsto x_1]$	by substitution
$\Gamma \vdash x_1 : A$	by SP-Var
$\Gamma \vdash x_2.l : A_2[L \mapsto x_1]$	by substitution

□

References

Andreas Abel and Dulma Rodriguez. 2008. Syntactic metatheory of higher-order subtyping. In *International Workshop on Computer Science Logic (CSL '08)*. 446–460.

Beniamino Accattoli and Giulio Guerrieri. 2016. Open Call-by-Value. In *Programming Languages and Systems*, Atsushi Igarashi (Ed.). Springer International Publishing, Cham, 206–226.

Wilhelm Ackermann. 1928. Zum hilbertschen aufbau der reellen zahlen. *Math. Ann.* 99, 1 (1928), 118–133.

Robin Adams. 2006. Pure type systems with judgemental equality. *Journal of Functional Programming* 16, 02 (2006), 219–246.

Robin Adams. 2008. A Survey of Predicativity. (2008). https://www.cs.ru.nl/R.Adams/predicativity3.pdf

Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. 2010. ΠΣ: Dependent Types without the Sugar. In *Functional and Logic Programming*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–55.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The essence of dependent object types. In *A List of Successes That Can Change the World*. Springer, 249–272.

Nada Amin, Adriaan Moors, and Martin Odersky. 2012a. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages (FOOL '12)*.

Nada Amin, Adriaan Moors, and Martin Odersky. 2012b. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages (FOOL '12)*. ACM.

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *OOPSLA '14*. ACM, 233–249.

David Aspinall and Adriana Compagnoni. 1996. Subtyping dependent types. In *LICS '96*. 86–97.

Lennart Augustsson. 1998. Cayenne — a Language with Dependent Types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/289423.289451

Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (1991), 125–154.

Henk Barendregt. 1992. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, Vol. 2. 117–309.

Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions* (1st ed.). Springer Publishing Company, Incorporated.

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2017. Type-preserving CPS Translation of Σ and Π Types is Not Not Possible. *Proc. ACM Program. Lang.* 2, POPL, Article 22 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158110

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. ACM, New York, NY, USA, 183–200. https://doi.org/10.1145/286936.286957

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.

Edwin C. Brady. 2011. IDRIS — Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification (PLPV '11)*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/1929529.1929536

Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. 1999. Comparing object encodings. *Information and Computation* 155, 1-2 (1999), 108–133.

Joana Campos and Vasco T. Vasconcelos. 2015. Imperative objects with dependent types. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs (FTfJP '15)*. ACM, 2:1–2:6.

Joana Campos and Vasco T. Vasconcelos. 2018. Dependent Types for Class-based Mutable Objects. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:28. https://doi.org/10.4230/LIPIcs.ECOOP.2018.13

Luca Cardelli. 1986a. Amber. In *Combinators and Functional Programming Languages*. Springer Berlin Heidelberg, 21–47.

Luca Cardelli. 1986b. *A Polymorphic lambda-calculus with Type: Type*. Digital Systems Research Center.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1994. An extension of system F with subtyping. *Information and Computation* 109, 1-2 (1994), 4–56.

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 471–523.

Chris Casinghino. 2014. *Combining Proofs and Programs*. Ph.D. Dissertation. The University of Pennsylvania.

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 33–45. https://doi.org/10.1145/2535838.2535883

Giuseppe Castagna and Gang Chen. 2001. Dependent types with subtyping and late-bound overloading. *Information and Computation* 168, 1 (2001), 1–67.

Gang Chen. 1997. Subtyping calculus of construction. *Mathematical Foundations of Computer Science 1997* (1997), 189–198.

Gang Chen. 1998. Dependent type system with subtyping (I) type level transitivity elimination. *Journal of Computer Science and Technology* 13, 6 (1998), 564–578.

Gang Chen. 2003. Coercive Subtyping for the Calculus of Constructions. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, 150–159.

James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types.* Technical Report CUCIS TR2003-1901.

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

Adriana Compagnoni and Healfdene Goguen. 2003. Typed operational semantics for higher-order subtyping. *Information and Computation* 184, 2 (2003), 242–297.

Adriana Beatriz Compagnoni. 1995. *Higher-order subtyping with intersection types.* Ph.D. Dissertation. University of Nijmegen.

Thierry Coquand. 1986. *An analysis of Girard's paradox.* Ph.D. Dissertation. INRIA.

Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76 (1988), 95–120.

Judicaël Courant. 1997. A module calculus for pure type systems. In *Typed Lambda Calculi and Applications*, Philippe de Groote and J. Roger Hindley (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 112–128.

Judicaël Courant. 2007. A module calculus for Pure Type Systems. *Journal of Functional Programming* 17, 3 (2007), 287–352.

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99).* ACM, New York, NY, USA, 50–63. https://doi.org/10.1145/301618.301641

Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of subsumption, minimum typing and type-checking in F$\leqslant$. *Mathematical structures in computer science* 2, 01 (1992), 55–91.

Derek Dreyer. 2005. *Understanding and Evolving the ML Module System.* Ph.D. Dissertation. Carnegie Mellon University.

Richard A. Eisenberg. 2016. *Dependent types in haskell: Theory and practice.* Ph.D. Dissertation. University of Pennsylvania.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14).* ACM, New York, NY, USA, 671–683. https://doi.org/10.1145/2535838.2535856

Leonidas Fegaras and Tim Sheard. 1996. Revisiting Catamorphisms over Datatypes with Embedded Functions (or, Programs from Outer Space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96).* ACM, New York, NY, USA, 284–294. https://doi.org/10.1145/237721.237792

Herman Geuvers. 1995. A short and flexible proof of strong normalization for the calculus of constructions. In *Types for Proofs and Programs*, Peter Dybjer, Bengt Nordström, and Jan Smith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 14–38.

Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation. Université Paris VII.

Brian Goetz. 2016. *JEP 286: Local-Variable Type Inference.* http://openjdk.java.net/jeps/286

James Gosling, Bill Joy, and Guy L. Steele. 1996. *The Java Language Specification* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Adam Michael Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. Dissertation. University of Strathclyde.

Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. 1994. Type classes in Haskell. In *Programming Languages and Systems — ESOP '94*, Donald Sannella (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 241–256.

Robert Harper. 2013. *Practical foundations for programming languages* (1st ed.). Cambridge University Press.

Robert Harper and Mark Lillibridge. 1994. A Type-theoretic Approach to Higher-order Modules with Sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 123–137. https://doi.org/10.1145/174675.176927

Robert Harper and John C. Mitchell. 1993. On the Type Structure of Standard ML. *ACM Trans. Program. Lang. Syst.* 15, 2 (April 1993), 211–252. https://doi.org/10.1145/169701.169696

James G. Hook and Douglas J. Howe. 1986. *Impredicative Strong Existential Equivalent to Type:Type.* Technical Report. Ithaca, NY, USA.

William A. Howard. 1980. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), 479–490.

DeLesley S. Hutchins. 2010. Pure Subtype Systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, 287–298.

Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. 1999. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. ACM, New York, NY, USA, 132–146. https://doi.org/10.1145/320384.320395

Mark P. Jones. 1993. A System of Constructor Classes: Overloading and Implicit Higher-order Polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, New York, NY, USA, 52–61. https://doi.org/10.1145/165180.165190

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Programming Languages and Systems*, Gert Smolka (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–244.

Jonas Kaiser, Tobias Tebbi, and Gert Smolka. 2017. Equivalence of System F and $\lambda 2$ in Coq Based on Context Morphism Lemmas. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP '17)*. ACM, 222–234.

Garrin Kimmell, Aaron Stump, Harley D. Eades, III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. 2012. Equational Reasoning About Programs with General Recursion and Call-by-value Semantics. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification (PLPV '12)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2103776.2103780

Xavier Leroy. 1990. *The ZINC Experiment: An Economical Implementation of the ML Language.* Technical Report 117. INRIA. http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-ZINC.html

Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 109–122. https://doi.org/10.1145/174675.176926

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2018. *The OCaml system release 4.07: Documentation and user's manual.* http://caml.inria.fr/pub/docs/manual-ocaml/ Release 4.07.

Mark Lillibridge. 1997. *Translucent sums: A foundation for higher-order module systems.* Ph.D. Dissertation. Carnegie Mellon University.

Yong Luo and Zhaohui Luo. 2004. Combining Incoherent Coercions for Σ -Types. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 276–292.

David B. MacQueen. 1986. Using Dependent Types to Express Modular Structure. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '86)*. ACM, New York, NY, USA, 277–286. https://doi.org/10.1145/512644.512670

Simon Marlow. 2010. *Haskell 2010 Language Report.* https://www.haskell.org/definition/haskell2010.pdf 2010.

Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)* (2010).

John Meacham. 2006. *Jhc User's Manual.* http://repetae.net/computer/jhc/manual.html

Microsoft Corporation. 2016. *TypeScript Language Specification.* https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md Version 1.8.

Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Trans. Program. Lang. Syst.* 10, 3 (July 1988), 470–502. https://doi.org/10.1145/44501.45065

T. Æ. Mogensen. 1992. Theoretical pearls: Efficient self-interpretation in lambda calculus. *Journal of Functional Programming* 2, 3 (1992), 345–364.

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Generics of a Higher Kind. In *OOPSLA '08*. ACM, 423–438.

Ulf Norell. 2007a. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Chalmers University of Technology.

Ulf Norell. 2007b. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Chalmers University of Technology.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An Overview of the Scala Programming Language.* Technical Report IC/2004/64. EPFL Lausanne, Switzerland.

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. 2003. A nominal theory of objects with dependent types. In *European Conference on Object-Oriented Programming (ECOOP '03)*. Springer, 201–224.

Luca Paolini and Simona Ronchi Della Rocca. 1999. Call-by-value Solvability. *RAIRO-Theoretical Informatics and Applications* 33, 6 (1999), 507–534.

Simon Peyton Jones and Erik Meijer. 1997. Henk: a Typed Intermediate Language. In *Types in Compilation Workshop*.

Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. 2004. *Wobbly types: type inference for generalised algebraic data types.* Technical Report MS-CIS-05-26. University of Pennsylvania.

F. Pfenning and C. Elliott. 1988. Higher-order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88).* ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/53990.54010

Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *Proceedings of the International Conference on Automated Deduction.* 202–206.

Benjamin C. Pierce. 1992. Bounded quantification is undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '92).* ACM, 305–315.

Benjamin C Pierce. 2002. *Types and programming languages.* MIT press.

Benjamin C. Pierce and Martin Steffen. 1997. Higher-order subtyping. *Theoretical computer science* 176, 1 (1997), 235–282.

Benjamin C. Pierce and David N. Turner. 1994. Simple type-theoretic foundations for object-oriented programming. *Journal of functional programming* 4, 02 (1994), 207–247.

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44.

Robert Pollack. 2002. Dependently Typed Records in Type Theory. *Formal Aspects of Computing* 13, 3 (01 Jul 2002), 386–402. https://doi.org/10.1007/s001650200018

Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. 2017. A Simple Soundness Proof for Dependent Object Types. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 46 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133870

John C. Reynolds. 1974. Towards a theory of type structure. In *Proceedings of the 'Colloque sur la Programmation'*. Paris, France, 408–425.

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16).* ACM, 624–641.

Andreas Rossberg. 2015. 1ML – Core and Modules United (F-ing First-class Modules). In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015).* ACM, New York, NY, USA, 35–47. https://doi.org/10.1145/2784731.2784738

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2010. F-ing Modules. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '10).* ACM, New York, NY, USA, 89–102. https://doi.org/10.1145/1708016.1708028

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*. Springer, 248–274.

David A. Schmidt. 1994. *The Structure of Typed Programming Languages.* MIT Press.

Paula G. Severi and Fer-Jan J. de Vries. 2012. Pure Type Systems with Corecursion on Streams: From Finite to Infinitary Normalisation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12).* ACM, New York, NY, USA, 141–152. https://doi.org/10.1145/2364527.2364550

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming*. 81–92.

Vincent Siles and Hugo Herbelin. 2012. Pure Type System conversion is always typable. *Journal of Functional Programming* 22, 2 (2012), 153–180.

Vilhelm Sjöberg. 2015. *A Dependently Typed Language with Nontermination.* Ph.D. Dissertation. University of Pennsylvania.

Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. 2012. Irrelevance, Heterogenous Equality, and Call-by-value Dependent Type Systems. In *Fourth workshop on Mathematically Structured Functional Programming (MSFP '12) (MSFP '12)*. 112–162.

Vilhelm Sjöberg and Stephanie Weirich. 2015. Programming Up to Congruence. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 369–382. https://doi.org/10.1145/2676726.2676974

Martin Steffen. 1998. *Polarized Higher-Order Subtyping*. Ph.D. Dissertation. Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg.

Sandro Stucki. 2017. *Higher-Order Subtyping with Type Intervals.* Ph.D. Dissertation. EPFL.

Aaron Stump. 2017. The calculus of dependent lambda eliminations. *Journal of Functional Programming* 27 (2017), e14. https://doi.org/10.1017/S0956796817000053

Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. 2008. Verified Programming in Guru. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification (PLPV '09)*. ACM, New York, NY, USA, 49–58. https://doi.org/10.1145/1481848.1481856

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '07)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 266–278. https://doi.org/10.1145/2034773.2034811

Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.

The Coq development team. 2016. *The Coq proof assistant reference manual.* https://coq.inria.fr/refman/ Version 8.6.

The GHC Team. 2018. *The Glasgow Haskell Compiler.* https://www.haskell.org/ghc/

The Rust Project Developers. 2011. *The Rust Programming Language.* https://doc.rust-lang.org/book/

Alan M Turing. 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society* 2, 1 (1937), 230–265.

L. S. van Benthem Jutting. 1993. Typing in pure type systems. *Information and Computation* 105, 1 (1993), 30–41.

Floris van Doorn, Herman Geuvers, and Freek Wiedijk. 2013. Explicit Convertibility Proofs in Pure Type Systems. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice (LFMTP '13)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2503887.2503890

Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. *PEP 484 – Type Hints*. https://www.python.org/dev/peps/pep-0484/

Philip Wadler. 1995. Monads for functional programming. In *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–52.

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 275–286. https://doi.org/10.1145/2500365.2500599

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* 115, 1 (1994), 38–94.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150

Hongwei Xi and Frank Pfenning. 1999. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '99)*. ACM, 214–227.

Yanpeng Yang, Xuan Bi, and Bruno C. d. S. Oliveira. 2016. Unified Syntax with Iso-types. In *Programming Languages and Systems*, Atsushi Igarashi (Ed.). Springer International Publishing, Cham, 251–270.

Yanpeng Yang and Bruno C. d. S. Oliveira. 2017. Unifying Typing and Subtyping. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 47 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133871

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/2103786.2103795

Jan Zwanenburg. 1999. Pure type systems with subtyping. In *TLCA '99*. 381–396.