

Unifying Typing and Subtyping

YANPENG YANG, The University of Hong Kong, Hong Kong, China

BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, Hong Kong, China

In recent years dependent types have become a hot topic in programming language research. A key reason why dependent types are interesting is that they allow unifying types and terms, which enables both additional *expressiveness* and *economy of concepts*. Unfortunately there has been much less work on dependently typed calculi for object-oriented programming. This is partly because it is widely acknowledged that the combination between dependent types and subtyping is particularly challenging.

This paper presents λI_{\leq} , which is a dependently typed generalization of System F_{\leq} . The resulting calculus follows the style of Pure Type Systems, and contains a single unified syntactic sort that accounts for both expressions, types and kinds. To address the challenges posed by the combination of dependent types and subtyping, λI_{\leq} employs a novel technique that unifies *typing* and *subtyping*. In λI_{\leq} there is only a 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. The resulting calculus has a rich metatheory and enjoys of several standard and desirable properties, such as *subject reduction*, *transitivity of subtyping*, *narrowing* as well as standard *substitution lemmas*. All the metatheory of λI_{\leq} is mechanically proved in the Coq theorem prover. Furthermore, (and as far as we are aware) λI_{\leq} is the first dependently typed calculus that completely subsumes System F_{\leq} , while preserving various desirable properties.

CCS Concepts: • **Theory of computation** → **Type theory**; • **Software and its engineering** → **General programming languages**;

Additional Key Words and Phrases: dependent types, subtyping

ACM Reference Format:

Yanpeng Yang and Bruno C. d. S. Oliveira. 2017. Unifying Typing and Subtyping. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 26 pages.
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 INTRODUCTION

Type systems for OOP languages are becoming increasingly more expressive and complex. For example the first versions of Java were simply typed. Java 5 introduced Generics, bringing (bounded) parametric polymorphism into mainstream OOP languages. Modern OOP languages, such as Scala, go further and include several advanced features such as *higher-order polymorphism* [Girard 1972; Moors et al. 2008] and *path-dependent types* [Odersky et al. 2004; Rompf and Amin 2016]. The extra complexity of the type systems is reflected in the significant effort to develop the corresponding metatheory. 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 [Rompf and Amin 2016]. DOT is an impressive feat, which required the development of several new proof techniques to prove type-safety and other properties.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2017 Copyright held by the owner/author(s).

2475-1421/2017/1-ART1

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

50 In recent years dependent types [Altenkirch et al. 2010; Augustsson 1998; Casinghino et al. 2014;
51 Coquand and Huet 1988; Sjöberg et al. 2012; Sjöberg and Weirich 2015; Stump et al. 2008; Weirich
52 et al. 2013] have become a hot topic in programming language research. In functional programming
53 dependent types are now trendy. Several new functional programming languages, most notably
54 Agda [Norell 2007] and Idris [Brady 2013], are now dependently typed. A key reason why dependent
55 types are interesting is that they naturally lead to a unification between types and terms, which
56 enables both additional *expressiveness* and *economy of concepts*. The added expressiveness comes
57 from the fact that types can now depend on values. Thus it becomes possible to express types
58 such as lists of a certain size n . Such sized list type ensures stronger invariants and is helpful
59 to prevent errors such as out-of-bounds errors. The other potential benefit of dependent types,
60 and the main motivator for our goals in this paper, is that once various different levels of syntax
61 (such as terms and types) are unified, then redundancy of language constructs at the various levels
62 can be avoided. This leads to an economy of concepts compared to more traditional calculi for
63 programming languages, which have different stratified levels of syntax. In turn, the economy of
64 concepts results in a significantly more compact metatheory, and can also lead to a reduction of
65 the necessary implementation effort. The key enabler for unifying terms and types in dependently
66 typed calculi is the adoption of a style similar to Pure Type Systems (PTSs) [Barendregt 1991]. In
67 PTSs there is only a single level of syntax for terms, and types (or kinds) that are expressed using
68 the same syntax. This is in contrast with more traditional calculi, where distinct pieces of syntax
69 (terms, types and kinds) are separated.

70 Like functional languages, OOP languages can also benefit from dependent types for exactly
71 the same reasons: added expressiveness; and economy of concepts. Given that the complexity
72 of type systems for OOP languages is so high, techniques for bringing down such complexity,
73 while retaining or even increasing expressiveness are certainly welcome. The economy of concepts
74 afforded by unified syntax typical of dependently typed languages can help here, since it can
75 significantly reduce the number of language constructs and relations needed in a calculus. Unfortu-
76 nately, there has been less work on dependently typed calculi for object-oriented programming. We
77 believe that there are essentially two primary reasons for this. The first reason, which applies to
78 programming languages in general (not just OOP), is that the interaction between general recursion
79 and dependent types is challenging. Essentially recursion breaks strong normalization, which many
80 common properties in dependently typed calculi depend upon. However, this area has been actively
81 investigated in the last few years, and a general approach [Kimmell et al. 2012; Sjöberg et al. 2012;
82 Sjöberg and Weirich 2015; Stump et al. 2008; Yang et al. 2016], based on explicit casts for type-level
83 computation, has emerged to provide an interesting solution for this problem. The second reason is
84 that how to smoothly combine dependent types and subtyping is still an open problem. Subtyping
85 is a substantial difference to traditional PTSs, which do not have such feature. The issue with
86 subtyping is well summarized by Aspinall and Compagnoni [1996]:

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

90
91 In essence the big difficulty is that the introduction of dependent types makes typing and
92 subtyping depend on each other. This causes several difficulties in developing the metatheory
93 for calculi that combine dependent types and subtyping. Practically all previous work [Aspinall
94 and Compagnoni 1996; Castagna and Chen 2001; Chen 1997, 2003; Zwanenburg 1999] attempts to
95 address such problem by somehow *untangling* typing and subtyping, which has the benefit that
96 the metatheory for subtyping can be developed before the metatheory of typing. Nevertheless,
97 several results and features prove to be challenging. For example, several systems [Aspinall and
98

99 Compagnoni 1996; Zwanenburg 1999] drop the support of *top types*, which are essential in OOP
100 programs to model the universal base class. Transitivity is difficult to prove as it may be entangled
101 with other properties such as subject reduction and strong normalization. Several studies [Aspinall
102 and Compagnoni 1996; Castagna and Chen 2001] have to use sophisticated techniques to show that
103 transitivity holds. *Pure Subtype Systems* [Hutchins 2010] take a different approach, by eliminating
104 typing and making subtyping the essential notion in the calculus. While this simplifies the syntax
105 and typing rules, and it is a very innovative idea, the metatheory is complex. Hutchins failed
106 to completely prove transitivity elimination and left several important lemmas that depend on
107 transitivity, such as subject reduction as conjectures instead. Finally, it is worthwhile mentioning
108 that we view the work on the DOT calculus as complementary to our own. The DOT calculus [Rompf
109 and Amin 2016] has path-dependent types, which are related but *different* to the dependent types
110 discussed in this paper; and it also has a very rich notion of bounds, that goes well beyond bounded
111 quantification. However, the DOT calculus does not attempt to unify types and terms, or typing
112 and subtyping. Also, the rich features of DOT prevent conventional properties such as *transitivity*
113 *elimination* (although it does have type-safety and an *axiomatic* transitivity property).

114 Despite the previous work on the combination of dependent types and subtyping, no calculi
115 has managed to subsume System F_{\leq} [Cardelli et al. 1994], together with its desirable properties
116 (for example transitivity elimination and subject reduction). System F_{\leq} is a standard polymorphic
117 calculus with subtyping, often identified as a canonical calculus capturing the essential OOP features
118 (and especially bounded quantification). Given the importance of System F_{\leq} as a foundational
119 model for OOP, it seems highly desirable that a dependently typed OOP calculus subsumes it.

120 This paper presents λI_{\leq} , which is a dependently typed generalization of System F_{\leq} . To address
121 the challenges posed by the combination of dependent types and subtyping, λI_{\leq} employs a novel
122 technique that unifies *typing* and *subtyping*. In λI_{\leq} there is only one judgment that is akin to a typed
123 version of subtyping. Both the typing relation, as well as type well-formedness are just special
124 cases of the subtyping relation. Therefore, λI_{\leq} takes a significantly different approach compared
125 to previous work. Previous work essentially attempts to fight the entanglement between typing
126 and subtyping. In contrast, what we propose with λI_{\leq} is to embrace such tangling, and essentially
127 collapsing the typing and subtyping relations into the same relation. This approach is different
128 from Hutchins' technique, which simply eliminates types and typing. λI_{\leq} retains types.

129 The λI_{\leq} calculus follows the style of Pure Type Systems, and contains a single unified syntactic
130 sort that accounts for both expressions, types and kinds. It is directly based on the λI calculus [Yang
131 et al. 2016], which is a dependently typed calculus with *iso-types*. Iso-types provide a simple form of
132 type casts, and λI_{\leq} adopts that idea to address the issues arising from the combination of recursion
133 and dependent types. The novelty over λI is the support for OOP features such as *higher-order*
134 *subtyping* [Pierce and Steffen 1997], *bounded quantification* and *top types*. To illustrate the expressive
135 power of λI_{\leq} , we show how object encodings relying on higher-order subtyping can be done in λI_{\leq} .
136 The resulting calculus enjoys several standard and desirable properties, such as *subject reduction*,
137 *transitivity of subtyping*, *narrowing* as well as standard *substitution lemmas*. All the metatheory of
138 λI_{\leq} and has been proved in the Coq theorem prover [The Coq development team 2016]. We also
139 provide an algorithmic version of λI_{\leq} based on bi-directional type-checking [Pierce and Turner
140 2000], which is shown to be sound and complete (also proved in Coq) with respect to the declarative
141 version. Finally we show that λI_{\leq} completely subsumes System F_{\leq} in expressive power. The manual
142 completeness proof of λI_{\leq} over System F_{\leq} is presented in the extended version [Yang and Oliveira
143 2017]. This proof is manual due to the well-known difficulties in mechanizing completeness proofs
144 between type systems with unified and stratified syntax [Kaiser et al. 2017].

145 In summary the contributions of this work are:
146
147

- 148 • **Unified subtyping:** A novel technique that unifies typing and subtyping into a single
149 relation. This technique enables the development of expressive dependently typed calculi
150 with subtyping.
- 151 • **The λI_{\leq} calculus:** A dependently typed calculus with subtyping that uses unified syntax, and
152 unified subtyping. The calculus supports *top types*, *higher-order polymorphism* and *bounded*
153 *quantification*. The paper presents a declarative version of the calculus, and a sound and
154 complete algorithmic version is discussed. A full specification of the algorithmic system is
155 presented in the extended version of this paper [Yang and Oliveira 2017].
- 156 • **Mechanized metatheory in Coq:** All proofs except for the completeness theorem over
157 System F_{\leq} have been mechanized and machine-checked in the Coq theorem prover¹.
- 158 • **Completeness of λI_{\leq} over System F_{\leq} :** A completeness proof showing that λI_{\leq} subsumes
159 System F_{\leq} is available in the extended version [Yang and Oliveira 2017].
- 160 • **Object encodings in λI_{\leq} :** As an example illustrating the expressive power of λI_{\leq} , we show
161 how object encodings relying on *higher-order subtyping*, and originally presented in System
162 F_{\leq}^{ω} [Pierce and Steffen 1997] can be done in λI_{\leq} .
163

164 2 OVERVIEW

165 In this section, we briefly introduce the concept of unified syntax and discuss the problem of com-
166 bining dependent types with subtyping. We informally introduce the key features of λI_{\leq} calculus,
167 namely unified subtyping and the support for dependent types by explicit casts. To illustrate the
168 suitability of λI_{\leq} to model objects, we adapt the existential object encoding [Bruce et al. 1999; Pierce
169 and Turner 1994] (originally based on System F_{\leq}^{ω}) to λI_{\leq} . The formal details of λI_{\leq} are further
170 discussed in Sections 3 and 4.
171

172 2.1 Unified Syntax versus Stratified Syntax

173 *Pure Type Systems* [Barendregt 1991] (PTSs) are a uniform framework for typed lambda calculi. It
174 features *unified* syntax which defines a single syntactic category for terms, types and kinds. This
175 brings economy in terms of syntax and defining relations over the system. In contrast, System
176 F_{ω} [Girard 1972], a higher-order lambda calculus, is usually presented using *stratified* syntax [Pierce
177 2002], which defines terms, types and kinds in distinct syntactic categories. System F_{\leq}^{ω} [Pierce
178 and Steffen 1997] extends System F_{ω} with subtyping and bounded quantification. Because of the
179 separation of syntax, the subtyping relation in System F_{\leq}^{ω} needs to be defined over multiple syntactic
180 forms of abstraction, i.e., abstraction over terms, types and type operators. This causes duplication
181 and complexity in the metatheory.

182 Note that System F_{ω} (without subtyping) can also be modeled with unified syntax: it is a special
183 case of PTSs and covered by Barendregt's λ -cube [Barendregt 1992]. It is tempting to adopt the
184 PTS-style unified syntax in System F_{\leq}^{ω} to simplify the subtyping relation. However, there are several
185 difficulties in applying such simplification to a higher-order system with bounded quantification.
186 Recall that there are three different forms of abstraction in System F_{\leq}^{ω} . It is hard to unify them
187 because the abstraction can quantify over a variable using two distinct relations, i.e., typing ($x : A$)
188 and subtyping ($X \leq A$):

189	Term abstraction	$\lambda x : A. e$
190	Type abstraction	$\lambda X \leq A. e$
191	Operator abstraction	$\lambda X \leq A. B$

192 To obtain a uniform representation of abstraction, we need to *unify the typing and subtyping relation*
193 in the first place. Moreover, calculi with PTS-style unified syntax usually allow *dependent types*, e.g.,
194

195 ¹Available from <https://bitbucket.org/yppyang/oopsla17>.
196

the calculus of constructions [Coquand and Huet 1988]. Combining dependent types and subtyping has its own problems, as will be discussed in the coming subsection.

2.2 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. The major reason is that allowing dependent types makes the typing and subtyping relations *entangled*. The subtyping and typing² judgments become mutually dependent. The typing judgment depends on subtyping because of the *subsumption* rule:

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}$$

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:

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

Circularity in the Metatheory. The mutual dependency causes circularity in the metatheory, since one cannot study properties of subtyping independently from typing. For example, λP_{\leq} [Aspinall and Compagnoni 1996] is an extension of the second-order dependently typed calculus λP [Barendregt 1992] with subtyping. In λP_{\leq} , the substitution lemmas for typing and subtyping depend on each other and require a more complicated proof by induction on four different judgments (i.e. subtyping, typing, kinding and formation) simultaneously. The transitivity of algorithmic subtyping requires types to be well-formed through beta-conversion. 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 λP_{\leq} . But such method is usually too specific and cannot be generally applied to other systems, e.g., the substitution proof in λP_{\leq} 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^{\leq} [Zwanenburg 1999], drop the support of top types because of such limitation.

2.3 Our Solution: Unified Subtyping

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

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

²Some 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.

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:

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

We solve the circularity problem because typing and subtyping cannot be mutually dependent in the first place — they are essentially the same relation. In λI_{\leq} , 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 λI_{\leq} . Moreover, the metatheory of λI_{\leq} is significantly simplified, e.g., there is only one form of substitution lemma to be proved, as discussed in Section 4.

Bounded quantification in λI_{\leq} . λI_{\leq} 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_{\leq}^{ω} , λI_{\leq} has a single form of abstraction: $\lambda x \leq e_1 : A. e_2$. By convention, the ordinary unbounded abstraction can be treated as syntactic sugar of a top-bounded one:

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

Notice that the top type (\top) is generalized to have any kind A instead of \star . With unified syntax, λI_{\leq} has fewer language constructs than System F_{\leq}^{ω} and a simpler definition of (unified) subtyping relation (see Section 3).

2.4 Type Casts: 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 so-called *conversion* rule, which allows beta equality between type expressions:

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

The rule checks the beta-equivalence of types and encounters evaluation, which terminates if both types are strongly normalizing. Thus, the decidability of type checking relies on strong normalization. For dependently typed languages with subtyping, the conversion rule is usually subsumed by the subsumption rule (see Section 2.2), which requires the subtyping relation $\Gamma \vdash A \leq B$ to subsume beta-equivalence $A =_{\beta} B$. Besides decidability, 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. Recently, several existing studies [Kimmell et al. 2012; Sjöberg et al. 2012; Sjöberg and Weirich 2015; Stump et al. 2008; 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.

λI_{\leq} adopts *iso-types* [Yang et al. 2016], which is one of the existing approaches to type casts. Iso-types can be viewed as a generalization of *iso-recursive types* [Crary et al. 1999; Pierce 2002], which works for arbitrary reducible terms rather than just (type-level) fixpoints. Two cast operators,

namely cast_\uparrow and cast_\downarrow , explicitly convert the type by one-step expansion and reduction, respectively. For example, if Int is the integer type and one-step reduction $(\lambda x : \star. x) \text{Int} \hookrightarrow \text{Int}$ holds, we have

$$\text{cast}_\uparrow [(\lambda x : \star. x) \text{Int}] \quad 3 : \text{Int} \quad e : (\lambda x : \star. x) \text{Int} \quad \text{cast}_\downarrow e : \text{Int}$$

Notice that the one-step reduction relation (\hookrightarrow) used in λI_{\leq} is *weak-head* and *call-by-name* (see Section 3.2). This makes the type conversion by casts less expressive than what is provided by the implicit conversion rule. For example, one cannot convert the length-indexed vector type $\text{Vec } (1 + 1)$ to $\text{Vec } 2$ by cast_\downarrow , since the desired reduction is not at the head position.

Nevertheless, we do not consider such loss of expressiveness problematic. The absence of conversion rule significantly simplifies the metatheory of λI_{\leq} 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. Cast operators are still expressive enough for our purposes: to model object encodings. Furthermore, there are alternative designs of casts which use full reduction to recover the expressiveness of the conversion rules, but they introduce some extra complications to the metatheory. Alternative approaches are discussed in Section 7.

2.5 Example: Object Encodings using λI_{\leq}

We show an example of object encodings in λI_{\leq} using the *existential encoding* method [Bruce et al. 1999; Pierce and Turner 1994] originally based on System F_{\leq}^{ω} . The encoding requires pairs, records and existential types which are not primitives but encodable in λI_{\leq} . We first show the encoding of *dependent sums* which generalize pairs and existential types before discussing object encodings.

Encoding Dependent Sums. Dependent sums are pairs where the second element can depend on the first one. The dependent sum type is also called a Sigma-type: $\Sigma x : A. B$, where x with type A can occur in B . In λI_{\leq} , dependent sum types can be encoded using *dependent function types* (i.e. Pi-types) in a similar way to Church-encoding existential types in System F or F_{\leq} [Pierce 2002]:

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

where **pack** and **unpack** are constructor and destructor of dependent sums, respectively. z is fresh such that $z \notin \text{FV}(\Pi x : A. B)$. Note that C is the type of e' . A and B can be derived from the type of e , i.e., $\Sigma x : A. B$. We can show that subtyping and typing rules of dependent sums are admissible in λI_{\leq} . The proof is trivial and available in the extended version [Yang and Oliveira 2017].

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:

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

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

Some OO concepts can be encoded in λI_{\leq} with dependent sums, e.g., type members in Scala [Odersky et al. 2004]. Consider an abstract interface of integer sets in Scala:

```

344 trait Set {
345   type T
346   def empty(): T
347   def member(x: Int, s: T): Boolean
348   def insert(x: Int, s: T): T
349 }
350
351
352

```

The type member T represents the abstract type of a set implementation. The interface contains three methods: `empty` returns an empty set; `member` checks if an element is in the set; and `insert` adds an element into the set. We can use dependent sums to encode `Set` in λI_{\leq} , assuming that we have primitive types (i.e. `Int` and `Bool`) and records (which can be encoded by pairs):

$$Set = \Sigma T : \star. \{empty : T, member : Int \rightarrow T \rightarrow Bool, insert : Int \rightarrow T \rightarrow T\}$$

A generic function f on `Set`, e.g.,

```

358   def f(s: Set) = s.member(3, s.insert(3, s.empty()))
359
360

```

can be encoded as follows:

$$f = \lambda s : Set. \mathbf{unpack} \ s \ \mathbf{as} \ [T, r] \ \mathbf{in} \ r.member \ 3 \ (r.insert \ 3 \ r.empty)$$

Notice that for simplicity reasons, we only encode 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 . It is non-trivial to Church-encode strong dependent sums without such restriction on `unpack` and using only Pi-types [Cardelli 1986b]. Nevertheless, weak dependent sums are sufficient for our purpose to encode existential types and non-dependent pairs and still more expressive. 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_{\leq} [Amin et al. 2016].

Encoding Objects. Now that pairs and existential types can be encoded in λI_{\leq} , we present the encoding of objects. Note again that records can be encoded with pairs using standard techniques [Pierce 2002] and that we assume λI_{\leq} is extended with integers, pairs, records and existential types in the following text. The existential encoding of objects [Pierce and Turner 1994] is as follows:

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

Obj is a type operator, i.e., a type-level function. The binder I denotes the interface. The body is an existential type which packs a pair. The pair consists of a hidden state (with type X) and methods which are functions depending on the state (with type $X \rightarrow I X$). For a concrete example of objects, we use the interface of cell objects [Bruce et al. 1999]:

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

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:

```

388   c = cast $\uparrow$ [Obj Cell] pack [{x : Int}, ({x = 0},  $\lambda s : \{x : Int\}. \text{cast}\uparrow$  [Cell {x : Int}]
389   {get = s.x, set =  $\lambda n : Int. \{x = n\}, bump = \{x = s.x + 1\}$  }) ]
390   as CellT
391
392

```


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 $Obj\ Cell$:

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

Note that we have two $cast_{\uparrow}$ operators here: one over the **pack** operator and another over the record of methods. Due to the lack of conversion rule in λI_{\leq} , 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. $CellT$). Noting that $Obj\ Cell \hookrightarrow CellT$, we can use $cast_{\uparrow}$ to do one-step type expansion for the package. Similarly, the second $cast_{\uparrow}$ operator in the definition of methods converts the record type into $Cell\ \{x : Int\}$. We use the following syntactic sugar for consecutive $cast_{\uparrow}$ and **pack**:

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

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

We 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 $getM$:

$$\begin{aligned} c \leftarrow get &\triangleq getM\ Cell\ c \\ getM &= \lambda I \leq Cell : \star \rightarrow \star. \lambda o : Obj\ I. \\ &\quad \mathbf{unpack}\ (\mathbf{cast}_{\downarrow}\ o)\ \mathbf{as}\ [X, (s, m)]\ \mathbf{in}\ (\mathbf{cast}_{\downarrow}\ (m\ s)).get \end{aligned}$$

$getM$ is parameterized by interface I and object o with such interface, where I can be any *sub-interface* of $Cell$. We first use the $cast_{\downarrow}$ operator to convert the type of o from $Obj\ I$ to the existential type $\exists X. X \times (X \rightarrow I\ X)$. Note that we extend the syntax of **unpack** with 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 $X \rightarrow I\ X$. The record of methods can be obtained by applying m to s . Noting that the subtyping relation $I\ X \leq Cell\ X$ holds, the type of $m\ s$ can be converted from $I\ X$ to $Cell\ X$ by subsumption. Another $cast_{\downarrow}$ 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:

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

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). For brevity, we omit the evaluation steps here which can be found in the extended version [Yang and Oliveira 2017]. The evaluation result is 1 as expected. We emphasize that the object encoding example exploits two fundamental features of λI_{\leq} , namely 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 THE λI_{\leq} CALCULUS

We present the λI_{\leq} 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]. The novelty over the λI calculus 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 λI_{\leq} in the rest of this section. Notice that λI_{\leq} discussed in this section does not contain recursion, which can be supported by following λI . We leave the discussion of recursion to Section 7.

3.1 Syntax

Figure 1 shows the syntax of expressions in λI_{\leq} . 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. Cast operators cast_{\uparrow} and cast_{\downarrow} (pronounced as “cast up” and “cast down”) are used for explicit type-level computation. Types that can be converted by cast operators are also called *iso-types* [Yang et al. 2016]. The cast operators were introduced in the λI calculus as a generalization of iso-recursive types. Similarly to the fold and unfold operators in iso-recursive types [Crary et al. 1999; Pierce 2002], cast_{\downarrow} and cast_{\uparrow} convert the type of an expression by a *one-step* reduction or expansion, respectively. cast_{\uparrow} needs to be annotated with the result type of one-step expansion, while cast_{\downarrow} does not, since one-step reduction is deterministic (see Section 4.4).

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

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., $\lambda x \leq \top : A. e$ and $\Pi x \leq \top : A. B$ as shown in Figure 1. We also treat arrow types $A \rightarrow B$ as syntactic sugar of $\Pi x : A. B$ if x does not occur free in B .

Context. The syntax of context Γ is defined in Figure 1. The variable binding only has the bounded form $x \leq e : A$ 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., $\Gamma, x : A \triangleq \Gamma, x \leq \top : A$.

3.2 Operational Semantics

Figure 2 shows the definition of one-step reduction (\leftrightarrow), 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 cast_{\downarrow} and cast_{\uparrow} . R-APP and R-CASTDN perform reduction at the head term of an application and the inner term of cast_{\downarrow} , respectively.

Since the reduction relation (\leftrightarrow) 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 \leftrightarrow . For example, an application starting with an variable: $x e_1 e_2 \dots e_n$. Also, as the top type is generalized,

491	Expressions	e, A, B	$::=$	$x \mid \star \mid \top \mid e_1 e_2 \mid \text{cast}_\uparrow[A] e \mid \text{cast}_\downarrow e$
492				$\mid \lambda x \leq e_1 : A. e_2 \mid \Pi x \leq e : A. B$
493	Contexts	Γ	$::=$	$\emptyset \mid \Gamma, x \leq e : A$
494	Inert Terms	u	$::=$	$x \mid \top \mid u e \mid \text{cast}_\downarrow u$
495	Values	v	$::=$	$\star \mid \lambda x \leq e_1 : A. e_2 \mid \Pi x \leq e : A. B \mid \text{cast}_\uparrow[A] e \mid u$
496	Syntactic Sugar	$\lambda x : A. e$	\triangleq	$\lambda x \leq \top : A. e$
497		$\Pi x : A. B$	\triangleq	$\Pi x \leq \top : A. B$
498		$A \rightarrow B$	\triangleq	$\Pi x : A. B$ where $x \notin \text{FV}(B)$

Fig. 1. Syntax

$e_1 \hookrightarrow e_2$

 Weak-head Reduction

R-BETA	R-APP	R-CASTDN	R-CASTELIM
$(\lambda x \leq e_3 : A. e_1) e_2 \hookrightarrow e_1[x \mapsto e_2]$	$e_1 \hookrightarrow e'_1$ $e_1 e_2 \hookrightarrow e'_1 e_2$	$e_1 \hookrightarrow e'_1$ $\text{cast}_\downarrow e_1 \hookrightarrow \text{cast}_\downarrow e'_1$	$\text{cast}_\downarrow (\text{cast}_\uparrow[A] e) \hookrightarrow e$

Fig. 2. Operational Semantics

assuming it is a supertype of an n -ary function, we can have a well-formed but stuck term such as $\top e_1 e_2 \dots e_n$. Furthermore, if we replace x and \top in both stuck terms by $\text{cast}_\downarrow x$ and $\text{cast}_\downarrow \top$ 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 [Accattoli and Guerrieri 2016; Paolini and Della Rocca 1999]. Figure 1 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., $\text{cast}_\downarrow u$. We treat inert terms as values. Figure 1 shows the syntax of values, ranged over by metavariable v , as shown in Figure 1. A value can either be the kind \star , a function, a function type, a 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 $\text{cast}_\uparrow[A] v$ as a value [Pierce 2002; Yang et al. 2016]. We will discuss these designs and their trade-offs later in Section 7.

3.3 Static Semantics

Figure 3 shows the rules of static semantics, including two judgment forms: context well-formedness $\vdash \Gamma$ and unified subtyping $\Gamma \vdash e_1 \leq e_2 : A$. The unified subtyping judgment $\Gamma \vdash e_1 \leq e_2 : A$ 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 sugar $\Gamma \vdash e : A$ (see Figure 3), 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 λI_{\leq} , subtyping, typing and well-formedness of types are all unified by the unified subtyping judgment:

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

540	$\vdash \Gamma$	Context Well-formedness		W-EMPTY
541				$\frac{\Gamma \vdash e : A \quad \Gamma \vdash A : \star}{\vdash \Gamma, x \leq e : A}$
542			$\frac{}{\vdash \emptyset}$	
543	$\Gamma \vdash e_1 \leq e_2 : A$	Unified Subtyping		
544				
545	S-AX	S-VARREFL	S-VARTRANS	S-TOP
546	$\frac{}{\vdash \Gamma}$	$\frac{}{\vdash \Gamma \quad x \leq e : A \in \Gamma}$	$\frac{}{x \leq e_1 : A \in \Gamma \quad \Gamma \vdash e_1 \leq e_2 : A}$	$\frac{}{\Gamma \vdash e : A}$
547	$\frac{}{\Gamma \vdash \star \leq \star : \star}$	$\frac{}{\Gamma \vdash x \leq x : A}$	$\frac{}{\Gamma \vdash x \leq e_2 : A}$	$\frac{}{\Gamma \vdash e \leq \top : A}$
548		S-ABS		
549	S-TOPREFL	S-PROD		
550	$\frac{}{\Gamma \vdash A : \star}$	$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash A : \star}{\Gamma, x \leq e_1 : A \vdash e_2 \leq e'_2 : B}$	$\frac{\Gamma, x \leq e_1 : A \vdash B : \star}{\Gamma, x \leq e_1 : A \vdash B : \star}$	
551	$\frac{}{\Gamma \vdash \top \leq \top : A}$	$\frac{}{\Gamma \vdash (\lambda x \leq e_1 : A. e_2) \leq (\lambda x \leq e_1 : A. e'_2) : \Pi x \leq e_1 : A. B}$		
552				
553	S-APP		S-CASTUP	S-CASTDN
554	$\frac{\Gamma \vdash e_1 \leq e_2 : \Pi x \leq e_3 : B. C \quad \Gamma \vdash A \leq e_3 : B}{\Gamma \vdash e_1 A \leq e_2 A : C[x \mapsto A]}$		$\frac{\Gamma \vdash B : \star \quad \Gamma \vdash e_1 \leq e_2 : A \quad B \hookrightarrow A}{\Gamma \vdash \text{cast}_{\uparrow}[B] e_1 \leq \text{cast}_{\uparrow}[B] e_2 : B}$	$\frac{\Gamma \vdash A' \leq A : \star \quad \Gamma \vdash e : A' \quad \Gamma \vdash A : \star}{\Gamma, x \leq e : A \vdash B : \star}$
555			$\frac{\Gamma, x \leq e : A \vdash B : \star \quad \Gamma, x \leq e : A' \vdash B \leq B' : \star}{\Gamma \vdash (\Pi x \leq e : A. B) \leq (\Pi x \leq e : A'. B') : \star}$	
556				
557			$\frac{\Gamma \vdash B : \star \quad \Gamma \vdash e_1 \leq e_2 : A \quad A \hookrightarrow B}{\Gamma \vdash \text{cast}_{\downarrow} e_1 \leq \text{cast}_{\downarrow} e_2 : B}$	
558				
559				
560		S-SUB		
561		$\frac{}{\Gamma \vdash e_1 \leq e_2 : A \quad \Gamma \vdash A \leq B : \star}$		
562		$\frac{}{\Gamma \vdash e_1 \leq e_2 : B}$		
563	Syntactic Sugar	$\Gamma \vdash e : A \triangleq \Gamma \vdash e \leq e : A$		

Fig. 3. Static Semantics

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 $x \leq e : A$ 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_{\leq} [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.

589 *Functions and Function Types.* S-ABS defines the relation between functions, which follows the
 590 *invariant* rule for type operators in System F_{\leq}^{ω} [Pierce and Steffen 1997]. It requires the bounds
 591 and argument types being compared to be identical. The first line of premises in S-ABS checks the
 592 well-formedness of binding. The second line of premises checks if the function bodies are covariant
 593 and their type is well-formed.

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

$$598 \frac{\Gamma \vdash A' \leq A \quad \Gamma \vdash B \leq B'}{\Gamma \vdash A \rightarrow B \leq A' \rightarrow B} \quad 599 \frac{\Gamma, x \leq e \vdash B \leq B'}{\Gamma \vdash \forall x \leq e. B \leq \forall x \leq e. B'}$$

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

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

611 *Explicit Casts and Syntactic Equality.* S-CASTUP and S-CASTDN are rules for explicit cast oper-
 612 ators. They can be seen as a generalization of typing rules of fold and unfold from iso-recursive
 613 types [Pierce 2002; Yang et al. 2016]. Weak-head reduction (\leftrightarrow) is used for type-level conversion.
 614 Note that when comparing cast_{\uparrow} terms, we require the annotations to be the same. S-SUB is the
 615 subsumption rule. The second premise checks the subtyping relation between well-formed types by
 616 reusing the unified subtyping judgment. Note that S-SUB does *not* subsume the implicit conversion
 617 rule, which can be found in F_{\leq}^{ω} and Pure Type Systems. Because the unified subtyping judgment
 618 does not subsume beta conversion, i.e., $(\lambda x : \star. x) \text{Int} \leq \text{Int}$ does not hold. As a consequence, types
 619 of expressions are equal only up to syntactic equality (i.e. alpha equality), but not beta equality.
 620 Nevertheless, we can recover type-level computation through cast operators in a similar way to
 621 iso-types [Yang et al. 2016].

622 *Algorithmic up to Subtyping.* The unified subtyping rules shown in Figure 3 are *declarative*
 623 because of the subsumption rule S-SUB. But the system is *almost* algorithmic: if we ignore the
 624 typing result and only consider the subtyping part, the system becomes algorithmic. Like the
 625 algorithmic version of System F_{\leq} , there is no built-in transitivity rule defined in λI_{\leq} . Actually,
 626 transitivity can be proved from other rules (see Section 4.2).

627 4 THE METATHEORY OF UNIFIED SUBTYPING

628 In this section, we discuss the metatheory of λI_{\leq} by focusing on two main targets: transitivity and
 629 type safety. We emphasize here that in previous work the metatheory for the combination between
 630 dependent types and subtyping was a key difficulty, greatly due to the entanglement between the
 631 metatheory of subtyping and typing. With unified subtyping we develop a single metatheory for
 632 the new relation instead. Traditional theorems related to the metatheory of typing and subtyping
 633 can then be viewed as particular instantiations of the unified subtyping theorems. Because the
 634

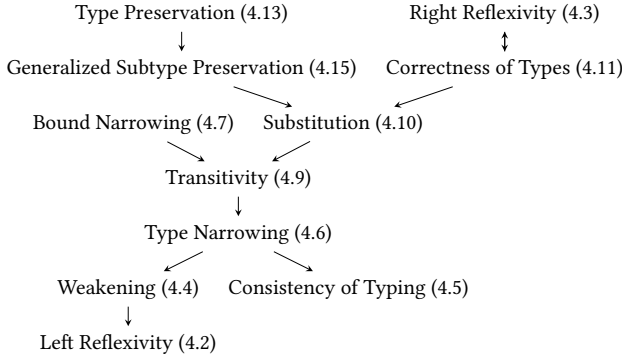


Fig. 4. Dependency of Lemmas for the Metatheory of Unified Subtyping

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 4. We only show the proof sketch and discuss interesting cases in this section. The full proofs (mechanized in Coq) can be found in the extended version [Yang and Oliveira 2017].

4.1 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_{\leq} is reflexive, i.e., $\Delta \vdash T \leq T$ holds for any *well-formed* type T and context Δ . Since unified subtyping in λU_{\leq} tracks typing results, the relation in reflexive form, i.e., $\Gamma \vdash e \leq e : A$, works like a typing judgment $\Gamma \vdash e : A$ (recall the syntactic sugar in Figure 3). Reflexivity does not hold for arbitrary e and A . However reflexivity does hold for any well-(sub)typed terms. That is:

LEMMA 4.1 (REFLEXIVITY). *If $\Gamma \vdash e_1 \leq e_2 : A$, then both $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$ hold.*

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 3.3. We separate the reflexivity lemma into two sub-lemmas by dividing the conclusion:

LEMMA 4.2 (LEFT REFLEXIVITY). *If $\Gamma \vdash e_1 \leq e_2 : A$, then $\Gamma \vdash e_1 : A$ holds.*

LEMMA 4.3 (RIGHT REFLEXIVITY). *If $\Gamma \vdash e_1 \leq e_2 : A$, then $\Gamma \vdash e_2 : A$ holds.*

Left reflexivity can be proved by induction on the derivation of $\Gamma \vdash e_1 \leq e_2 : A$. However, right reflexivity is difficult to prove due to the generalized top type. Consider the case of S-TOP, i.e., $\Gamma \vdash e \leq \top : A$. 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 4.11), which is not available currently. We will show the full proof later in Section 4.3. 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 $\Gamma \vdash A' \leq A : \star$ 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 4.4 (WEAKENING). *If $\Gamma_1, \Gamma_3 \vdash e_1 \leq e_2 : A$ and $\Gamma_1, \Gamma_2, \Gamma_3$, then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash e_1 \leq e_2 : A$.*

The proof is by induction on the derivation of $\Gamma_1, \Gamma_3 \vdash e_1 \leq e_2 : A$. The only interesting case is when S-PROD is the last derivation. The last premise of S-PROD adds binding $x \leq e : A'$ 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 4.2) to the first premise, i.e., $\Gamma \vdash A' \leq A : \star$.

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

LEMMA 4.5 (CONSISTENCY OF TYPING). *If $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_1 \leq e_2 : B$, then $\Gamma \vdash e_1 \leq e_2 : A$.*

The proof is by induction on the derivation of $\Gamma \vdash e_1 \leq e_2 : B$. This lemma is the key to decoupling typing from unified subtyping. To prove $\Gamma \vdash e_1 \leq e_2 : A$, 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 $\Gamma \vdash e_1 \leq e_2 : B$.

Narrowing. We have two narrowing lemmas in λ_{\leq} , type narrowing and bound narrowing:

LEMMA 4.6 (TYPE NARROWING). *Given $\Gamma_1, x \leq e : B, \Gamma_2 \vdash e_1 \leq e_2 : C$, if $\Gamma_1 \vdash A \leq B : \star$ and $\Gamma_1 \vdash e : A$, then $\Gamma_1, x \leq e : A, \Gamma_2 \vdash e_1 \leq e_2 : C$.*

LEMMA 4.7 (BOUND NARROWING). *If $\Gamma_1, x \leq e : B, \Gamma_2 \vdash e_1 \leq e_2 : C$ and $\Gamma_1 \vdash e' \leq e : B$, then $\Gamma_1, x \leq e' : B, \Gamma_2 \vdash e_1 \leq e_2 : C$.*

As indicated by the name, for a binding $x \leq e : B$ 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 4.3. The type narrowing lemma is proved by induction on the derivation of $\Gamma_1, x \leq e : B, \Gamma_2 \vdash e_1 \leq e_2 : C$. 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 $\Gamma_1, x \leq e : A, \Gamma_2 \vdash x \leq e_2 : B$. By applying the subsumption rule S-SUB and S-VARTRANS, our target becomes $\Gamma_1, x \leq e : A, \Gamma_2 \vdash e \leq e_2 : A$. Note that we have $\Gamma_1, x \leq e : A, \Gamma_2 \vdash e \leq e_2 : B$ 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 4.5) and prove $\Gamma_1, x \leq e : A, \Gamma_2 \vdash e : A$ instead, which is immediate by weakening (Lemma 4.4).

4.2 Transitivity

Transitivity is a desirable property of systems with subtyping. The declarative presentation often includes a built-in transitivity rule:

$$\frac{\Gamma \vdash e_1 \leq e_2 : A \quad \Gamma \vdash e_2 \leq e_3 : A}{\Gamma \vdash e_1 \leq e_3 : A} \text{S-TRANS}$$

It 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* [Compagnoni 1995; Pierce 2002; Pierce and Steffen 1997] 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 λI_{\leq} without a built-in transitivity rule but only with a base case for variables (i.e. S-VARTRANS), as mentioned in Section 3.3. Next we show the proof of transitivity in λI_{\leq} .

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:

$$\frac{\frac{\text{S-SUB} \quad \Gamma \vdash e_1 \leq e_2 : B_1 \quad \Gamma \vdash B_1 \leq A : \star}{\Gamma \vdash e_1 \leq e_2 : A} \quad \frac{\text{S-SUB} \quad \Gamma \vdash e_2 \leq e_3 : B_2 \quad \Gamma \vdash B_2 \leq A : \star}{\Gamma \vdash e_2 \leq e_3 : A}}{\Gamma \vdash e_1 \leq e_3 : A} \text{S-TRANS}$$

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

$$\frac{\Gamma \vdash e_1 \leq e_2 : A \quad \Gamma \vdash e_2 \leq e_3 : B}{\Gamma \vdash e_1 \leq e_3 : A} \text{S-TRANS2}$$

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 $\Gamma \vdash e_1 \leq e_2 : A$. We only discuss the interesting case when both derivations end with S-PROD. We have $e_1 = \Pi x \leq e : A_1. B_1$, $e_2 = \Pi x \leq e : A_2. B_2$, and $e_3 = \Pi x \leq e : A_3. B_3$, with

$$\begin{aligned} \Gamma \vdash A_2 \leq A_1 : \star & \quad (1) & \Gamma, x \leq e : A_2 \vdash B_1 \leq B_2 : \star & \quad (2) \\ \Gamma \vdash A_3 \leq A_2 : \star & \quad (3) & \Gamma, x \leq e : A_3 \vdash B_2 \leq B_3 : \star & \quad (4) \end{aligned}$$

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 $\Gamma \vdash A_3 \leq A_1 : \star$ and $\Gamma, x \leq e : A_3 \vdash B_1 \leq B_3 : \star$. 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 4.6 to narrow the type of the binding to obtain $\Gamma, x \leq e : A_3 \vdash B_1 \leq B_2 : \star$. 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 4.8 (GENERALIZED TRANSITIVITY). *If $\Gamma \vdash e_1 \leq e_2 : A$ and $\Gamma \vdash e_2 \leq e_3 : B$, then $\Gamma \vdash e_1 \leq e_3 : A$.*

Thus, the original transitivity is an immediate corollary:

LEMMA 4.9 (TRANSITIVITY). *If $\Gamma \vdash e_1 \leq e_2 : A$ and $\Gamma \vdash e_2 \leq e_3 : A$, then $\Gamma \vdash e_1 \leq e_3 : A$.*

As shown in Figure 4, the proof of generalized transitivity depends on type narrowing (Lemma 4.6) and type narrowing depends on consistency of typing (Lemma 4.5). 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.

4.3 Basic Lemmas, Revisited

Recall that in Section 4.1 we leave two lemmas unproved, i.e., right reflexivity (Lemma 4.3) and bound narrowing (Lemma 4.7), which depend on other lemmas that were not available yet. As we have proved transitivity in Section 4.2, we can recover the proof of these two lemmas.

Bound Narrowing. Similar to type narrowing (Lemma 4.6), bound narrowing (Lemma 4.7) is proved by induction on the derivation of $\Gamma_1, x \leq e : B, \Gamma_2 \vdash e_1 \leq e_2 : C$. 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 $\Gamma_1, x \leq e' : B, \Gamma_2 \vdash x \leq e_2 : B$. By the induction hypothesis, we have $\Gamma_1, x \leq e' : B, \Gamma_2 \vdash e \leq e_2 : B$. Noticing that by weakening (Lemma 4.4), we can obtain $\Gamma_1, x \leq e' : B, \Gamma_2 \vdash e' \leq e : B$ from the second condition. By transitivity (Lemma 4.9), we have $\Gamma_1, x \leq e' : B, \Gamma_2 \vdash e' \leq e_2 : B$. Also noticing that $x \leq e' : B \in \Gamma_1, x \leq e' : B, \Gamma_2$, we obtain the target by the rule S-VARTRANS.

Substitution. We show that the substitution lemma holds in λI_{\leq} :

LEMMA 4.10 (SUBSTITUTION). *If $\Gamma_1, x \leq e : B, \Gamma_2 \vdash e_1 \leq e_2 : A$ and $\Gamma_1 \vdash e' \leq e : B$, then $\Gamma_1, \Gamma_2[x \mapsto e'] \vdash e_1[x \mapsto e'] \leq e_2[x \mapsto e'] : A[x \mapsto e']$.*

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 $\Gamma_1 \vdash e' \leq e : B$ contains both subtyping requirement (e' is a subtype of e) and typing requirement (e' has type B). Thus, the substitution lemma in λI_{\leq} has only one form.

Right Reflexivity. As mentioned in Section 4.1, right reflexivity (Lemma 4.3) depends on correctness of types:

LEMMA 4.11 (CORRECTNESS OF TYPES). *If $\Gamma \vdash e_1 \leq e_2 : A$, then $\Gamma \vdash A : \star$.*

But correctness of types also depends on right reflexivity. Consider the last derivation of $\Gamma \vdash e_1 \leq e_2 : A$ is S-SUB, where the premises are $\Gamma \vdash e_1 \leq e_2 : B$ and $\Gamma \vdash B \leq A : \star$. 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 $\Gamma \vdash e_1 \leq e_2 : A$. Note that the proof of correctness of types also depends on the substitution lemma (Lemma 4.10) when the derivation ends with S-APP.

With both left and right reflexivity proved, we conclude the reflexivity (Lemma 4.1) holds and the interpretation of unified subtyping in Section 3.3 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 4).

4.4 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 3.3.

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

LEMMA 4.12 (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 cast_{\downarrow} term is not required to be annotated with the result type.

834 *Type Preservation.* Type preservation, also known as subject reduction [Wright and Felleisen
835 1994], states that reducing a term does not change its type:

836 LEMMA 4.13 (TYPE PRESERVATION). *If $\Gamma \vdash e : A$ and $e \hookrightarrow e'$, then $\Gamma \vdash e' : A$.*
837

838 However, if we try to directly prove this lemma by induction on the derivation of $\Gamma \vdash e \leq e' : A$
839 (i.e. $\Gamma \vdash e : A$), the proof will get stuck. Consider the last derivation is S-CASTDN and $e \hookrightarrow e'$ is
840 an instance of R-CASTELIM with $e = \text{cast}_\downarrow(\text{cast}_\uparrow[B'] e)$ and $e' = e$. We have $\Gamma \vdash \text{cast}_\uparrow[B'] e : B$
841 and $B \hookrightarrow A$. By inversion of S-CASTUP, we can obtain $\Gamma \vdash e : A'$, $B' \hookrightarrow A'$ and $\Gamma \vdash B' \leq B : \star$.
842 Our target is to show $\Gamma \vdash e : A$. If we can prove $\Gamma \vdash A' \leq A : \star$, then the target can be obtained
843 immediately by the subsumption rule S-SUB. The relation is shown as follows:

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

844 The subtyping relation in the second line requires a proof, which can be shown by the following
845 lemma with a more general typing result other than the kind \star :
846

847 LEMMA 4.14 (SUBTYPE PRESERVATION). *If $\Gamma \vdash e_1 \leq e_2 : A$, $e_1 \hookrightarrow e'_1$, $e_2 \hookrightarrow e'_2$, then $\Gamma \vdash e'_1 \leq e'_2 : A$.*
848

849 We call this lemma *subtype preservation* indicating that the unified subtyping relation is preserved
850 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
851 proof is by induction on the derivation of $\Gamma \vdash e_1 \leq e_2 : A$. The substitution lemma (Lemma 4.10)
852 is required for the case when the derivation ends with S-APP and both reductions are instances
853 of R-BETA. However, the proof gets stuck when the derivation ends with S-CASTDN, and both
854 reductions are instances of R-CASTELIM with $e_1 = \text{cast}_\downarrow(\text{cast}_\uparrow[B] e'_1)$ and $e_2 = \text{cast}_\downarrow(\text{cast}_\uparrow[B] e'_2)$.
855 The induction hypothesis does not work as it requires $\text{cast}_\uparrow[B] e'_1$ and $\text{cast}_\uparrow[B] e'_2$ to be reducible,
856 while both of them are values (see Figure 1). To solve this issue, we need to generalize the subtype
857 preservation lemma into the following one:
858

859 LEMMA 4.15 (GENERALIZED SUBTYPE PRESERVATION). *Given that $\Gamma \vdash e_1 \leq e_2 : A$ holds,*
860

- 861 (1) *if both e_1 and e_2 are cast_\uparrow terms, i.e., $e_1 = \text{cast}_\uparrow[B] e'_1$ and $e_2 = \text{cast}_\uparrow[B] e'_2$, and $A \hookrightarrow A'$,
862 $B \hookrightarrow B'$, then $\Gamma \vdash e'_1 \leq e'_2 : A'$;*
- 863 (2) *otherwise, if $e_1 \hookrightarrow e'_1$ and $e_2 \hookrightarrow e'_2$, then $\Gamma \vdash e'_1 \leq e'_2 : A$.*
864

865 Now the proof by induction can proceed with the generalized lemma. For the case which was stuck
866 in the previous attempt, the conclusion is exactly the induction hypothesis that follows the case (1)
867 of the lemma. The non-trivial case is when the derivation ends with the subsumption rule S-SUB.
868 When e_1 and e_2 are not both cast_\uparrow terms, the proof is trivial by the induction hypothesis. Otherwise,
869 we have $e_1 = \text{cast}_\uparrow[C] e'_1$ and $e_2 = \text{cast}_\uparrow[C] e'_2$ such that
870

$$\begin{array}{ll} \Gamma \vdash \text{cast}_\uparrow[C] e'_1 \leq \text{cast}_\uparrow[C] e'_2 : B & (1) \quad \Gamma \vdash B \leq A : \star & (2) \\ A \hookrightarrow A' & (3) \quad C \hookrightarrow C' & (4) \end{array}$$

871 Our target is to show $\Gamma \vdash e'_1 \leq e'_2 : A'$. Note that the annotations of cast_\uparrow in both terms must be the
872 same (i.e. C) by S-CASTUP. By inversion of (1), we have $\Gamma \vdash C \leq B : \star$. We first show there exists
873 some B' such that $B \hookrightarrow B'$ by proving the following lemma:
874

875 LEMMA 4.16 (REDUCTION EXISTS IN THE MIDDLE). *Given that $\Gamma \vdash C \leq B : D$ and $\Gamma \vdash B \leq A : D$, if
876 $C \hookrightarrow C'$ and $A \hookrightarrow A'$, then there exists B' such that $B \hookrightarrow B'$.*
877

878 Then by induction hypothesis, we have $\Gamma \vdash e'_1 \leq e'_2 : B'$ from (1) by the first case of lemma and
879 $\Gamma \vdash B' \leq A' : \star$ from (2) by the second (1st case impossible). Thus, we can prove the target by S-SUB
880

883 and conclude Lemma 4.15. Finally, it is trivial to show that the original subtype preservation lemma
 884 is a corollary of the generalized one. Thus, we can conclude the type preservation lemma which is
 885 an immediate corollary of subtype preservation.

886 *Progress.* Progress states that well-formed terms do not get stuck:
 887

888 LEMMA 4.17 (PROGRESS). *If $\emptyset \vdash e : A$ then either e is a value v or there exists e' such that $e \multimap e'$.*

889 As we mentioned in Section 3.2, the type-level reduction in cast operators may encounter open
 890 terms. We prove a stronger progress lemma with a non-empty context:

891 LEMMA 4.18 (GENERALIZED PROGRESS). *If $\Gamma \vdash e : A$ then either e is a value v or there exists e' such
 892 that $e \multimap e'$.*

893
 894 Then the original progress lemma is an immediate corollary of the stronger version. The proof is
 895 straightforward by induction on the derivation of $\Gamma \vdash e : A$. The definition of values is critical to
 896 the proof as it covers many stuck terms with variables and the top type (see also the discussion of
 897 inert terms in Section 3.2).

899 5 ALGORITHMIC VERSION

900 As we mentioned in Section 3.3, the unified subtyping judgment presented in Figure 3 is declarative
 901 but almost algorithmic. The typing part is declarative because of the subsumption rule, while
 902 the subtyping part is algorithmic. If we separately check the typing part and subtyping part, we
 903 just need to develop an algorithm for type checking. We use *bidirectional type checking* [Pierce
 904 and Turner 2000], a standard technique to develop the type checking algorithm for type systems
 905 with subtyping. We only briefly introduce the bidirectional system without showing rules which
 906 are mostly standard. For space reasons, the full specification is omitted and can be found in the
 907 extended version [Yang and Oliveira 2017]. We show the soundness and completeness of the type
 908 and subtype checking algorithm with respect to the original unified subtyping judgment. The full
 909 proof (mechanized in Coq) is available in the extended version [Yang and Oliveira 2017]. Developing
 910 a unified algorithmic system is left as future work, as will be discussed in Section 7.

912 5.1 Bidirectional Type Checking

913 We extend the syntax of λI_{\leq} with annotations, denoted by $(e : A)$ (parentheses are required). We
 914 use $|e|$ to denote the erasure of all annotation from a term and $|\Gamma|$ for the erasure of a context. The
 915 algorithmic subtyping judgment is denoted by $\Gamma \vdash e_1 \leq e_2$. It is developed by removing the typing
 916 part of unified subtyping rules in Figure 3.

917 The algorithmic typing judgment has two directions: the checking judgment $\Gamma \vdash e \Leftarrow A$ and
 918 the synthesis judgment $\Gamma \vdash e \Rightarrow A$. They are developed by following the typing part of unified
 919 subtyping. Most syntactic forms are typed by the synthesis judgment, including functions and
 920 function types since both binders are annotated. Two syntactic forms that are not annotated require
 921 the checking judgment, namely the top type (\top) and cast_{\downarrow} term. The subsumption rule from the
 922 unified subtyping is adapted to the checking direction.

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

931

932	Types	T	$::=$	$X \mid \top \mid T_1 \rightarrow T_2 \mid \forall X \leq T_1. T_2$		
933	Terms	t	$::=$	$x \mid \lambda x : T. t \mid t_1 t_2 \mid \Lambda X \leq T. t \mid t[T]$		
934	Contexts	Δ	$::=$	$\emptyset \mid \Delta, x : T \mid \Delta, X \leq T$		
935						
936	$T^* = A$	Mapping of Type		$t^* = e$	Mapping of Term	$\Delta^* = \Gamma$
937	$\top^* = \top$			$x^* = x$		$\emptyset^* = \emptyset$
938	$X^* = X$			$(\lambda x : T. t)^* = \lambda x \leq \top : T^*. t^*$		$(\Delta, x : T)^* = \Delta^*, x \leq \top : T^*$
939	$(T_1 \rightarrow T_2)^* = \Pi x \leq \top : T_1^*. T_2^*$			$(t_1 t_2)^* = t_1^* t_2^*$		$(\Delta, X \leq T)^* = \Delta^*, X \leq T^* : \star$
940	$(\forall X \leq T_1. T_2)^* = \Pi X \leq T_1^* : \star. T_2^*$	$(x \text{ Fresh})$		$(\Lambda X \leq T. t)^* = \lambda X \leq T^* : \star. t^*$		
941				$(t[T])^* = t^* T^*$		
942						

Fig. 5. Syntax and Translation of System F_{\leq}

5.2 Soundness and Completeness

We show that the algorithmic subtyping and typing are both sound and complete to the original unified subtyping. We use $\Gamma \vdash e \Leftarrow A$ 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 5.1 (SOUNDNESS OF ALGORITHM). *If $\Gamma \vdash e_1 \Leftarrow A$, $\Gamma \vdash e_2 \Leftarrow A$ and $\Gamma \vdash e_1 \leq e_2$, then $|\Gamma| \vdash |e_1| \leq |e_2| : |A|$.*

THEOREM 5.2 (COMPLETENESS OF ALGORITHM). *If $\Gamma \vdash e_1 \leq e_2 : A$, then $\Gamma \vdash e_1 \leq e_2$ 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 $|e'_2| = e_2$.*

6 SUBSUMPTION OF SYSTEM F_{\leq}

λI_{\leq} is a generalization of System F_{\leq} with dependent types. In this section, we show that λI_{\leq} can completely subsume the *Kernel Fun* variant [Cardelli and Wegner 1985] of System F_{\leq} . We first show the translation from System F_{\leq} to λI_{\leq} and prove that the typing and subtyping judgments of System F_{\leq} still hold in λI_{\leq} up to mapping. The full proofs and specification of System F_{\leq} are available in the extended version of this paper [Yang and Oliveira 2017].

6.1 Translating System F_{\leq} to λI_{\leq}

We show the syntax of System F_{\leq} and mapping (denoted by $*$) of types, terms and contexts from System F_{\leq} to λI_{\leq} in Figure 5. We use the metavariable T for types, t for terms and Δ for contexts in System F_{\leq} . 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 1. 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.

6.2 Subsumption of Typing and Subtyping

We prove that the mapped typing and subtyping relations still hold in λI_{\leq} . The type system of System F_{\leq} 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 6.1 (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:

981 THEOREM 6.2 (SUBSUMPTION OF SUBTYPING). *If $\Delta \vdash T_1 \leq T_2$, then $\Delta^* \vdash T_1^* \leq T_2^* : \star$.*

982 THEOREM 6.3 (SUBSUMPTION OF TYPING). *If $\Delta \vdash t : T$, then $\Delta^* \vdash t^* : T^*$.*

983
984 The proof is straightforward by induction on the derivation of subtyping relation $\Delta \vdash T_1 \leq T_2$
985 and typing relation $\Delta \vdash t : T$, respectively. Note that the mapped typing relation $\Delta^* \vdash t^* : T^*$ is
986 syntactic sugar of unified subtyping relation, i.e., $\Delta^* \vdash t^* \leq t^* : T^*$ (see Figure 1).
987

988 7 DISCUSSION

989 In this section, we discuss alternative designs for λI_{\leq} and justify their trade-offs to the current
990 design.
991

992 *Recursion and Recursive Types.* The current syntax of λI_{\leq} does not contain any form of recursion.
993 Adding recursion and recursive types is easy by simply following the treatment of recursion in
994 λI [Yang et al. 2016]. We have an alternative formulation of our system (including full proofs) with
995 those features. However subtyping recursive types reveals an interesting problem. The typical
996 Amber rule [Cardelli 1986a], or even the following restricted invariant rule

$$997 \frac{\Gamma, x \leq \top : A \vdash e_1 \leq e_2 : A \quad \Gamma \vdash A : \star}{\Gamma \vdash (\mu x : A. e_1) \leq (\mu x : A. e_2) : A} \text{S-MuI}$$

1000 does not work well with λI_{\leq} . Here $\mu x : A. e_1$ is a recursive type with the recursive binder x that
1001 can appear in the body e_1 . The rule requires the types of recursive binders to be the same. We add a
1002 new reduction rule to unroll a recursive type: $\mu x : A. e \hookrightarrow e[x \mapsto \mu x : A. e]$. In order to keep type
1003 soundness, we need to ensure subtype preservation (Lemma 4.14) still holds. If $f = \lambda y : \star. y$ is an
1004 identity type operator with type $\star \rightarrow \star$, consider
1005

$$1006 \mu x : \star. f x \leq \mu x : \star. \top x$$

1007 This relation holds by the rule S-MuI because we have $f \leq \top : \star \rightarrow \star$ by S-Top and then
1008 $x : \star \vdash f x \leq \top x : \star$ by S-APP. Subtype preservation requires that the subtyping relation still
1009 holds with both sides reduced by one step:
1010

$$1011 f (\mu x : \star. f x) \leq \top (\mu x : \star. \top x) \quad (1)$$

1012 However, (1) does not hold because the pointwise subtyping rule S-APP requires arguments of
1013 two applications should be the same. Thus, types are not preserved using the invariant rule for
1014 subtyping recursive types. This issue appears to be common to most systems with *higher-order*
1015 *subtyping* [Aspinall and Compagnoni 1996; Pierce and Steffen 1997; Zwanenburg 1999], as it arises
1016 from the interaction between the rules for recursive types and rules that use pointwise subtyping.
1017

1018 To solve this issue, we either change the S-APP rule to be polarized [Steffen 1998], or only allow
1019 subtyping two identical recursive types. The former approach is interesting, but requires a major
1020 modification to the system. We leave that approach for future work. The latter approach is relatively
1021 simple by using the following rule:

$$1022 \frac{\Gamma, x \leq \top : A \vdash e : A \quad \Gamma \vdash A : \star}{\Gamma \vdash (\mu x : A. e) \leq (\mu x : A. e) : A} \text{S-Mu}$$

1023
1024 Due to the unified syntax, $\mu x : A. e$ can serve as both the term-level fixpoint and recursive type.
1025 Though full subtyping of recursive types is not possible in λI_{\leq} currently, we are still able to introduce
1026 general recursion and recursive types to the system with S-Mu. This is precisely the approach used
1027 in our alternative formulation.
1028
1029

1030 *Operational Semantics.* λI_{\leq} uses the same call-by-name (CBN) operational semantics that λI [Yang
 1031 et al. 2016] uses. However most OO languages use call-by-value (CBV). CBV semantics is more
 1032 complicated because of the existence of dependent types and explicit casts in λI_{\leq} , but we believe
 1033 that it should also be possible to have a variant of the calculus with CBV. We also treat $\text{cast}_{\uparrow}[A] e$
 1034 as a value (see Section 3.2), which follows the standard call-by-name semantics of iso-recursive
 1035 types [Harper 2013]. Such design makes the cast_{\uparrow} operator *computationally relevant*. Alternatively,
 1036 we can take the approach from λI , which treats $\text{cast}_{\uparrow}[A] v$ as a value and adds a reduction rule to
 1037 further reduce the inner term of cast_{\uparrow} . However, the alternative semantics of cast_{\uparrow} leads to more
 1038 complex reduction rules and metatheory. The cast canceling rule R-CASTELIM (See Figure 2) now
 1039 needs to check if the inner term of cast_{\uparrow} is a value, which requires some non-trivial changes to
 1040 current proofs of the metatheory. We leave the CBV semantics and computational irrelevance of
 1041 casts as future work.

1042 *Top Types.* For top types, we can alternatively treat only \top as a value but not $\top e_1 \dots e_n$, which
 1043 is an inert term (see Figure 1). In such design additional reduction rules similar to the $\beta\top$ -reduction
 1044 rules of System F_{\leq}^{ω} [Pierce and Steffen 1997] are needed to further reduce “stuck” terms to values,
 1045 i.e., $\top e \hookrightarrow \top$. However, the approach of using $\beta\top$ -reduction needs to define reduction rules for
 1046 each form of stuck terms, e.g., $\top e$ and $\text{cast}_{\downarrow} \top$, while the definition of inert terms deals with stuck
 1047 terms in a more uniform way.

1048 *Weak vs Full Casts.* Cast operators in λI_{\leq} use the same weak-head reduction for type-level
 1049 computation. As mentioned in Section 2.4, certain type conversions cannot be performed by weak-
 1050 head reduction/expansion if they require reduction at non-head position, e.g., converting $\text{Vec } (1 + 1)$
 1051 to $\text{Vec } 2$. To address this limitation we can use an alternative design from the λI_p variant [Yang et al.
 1052 2016] of λI . In that design *full* reduction is used in cast operators, which allows reduction at any
 1053 position of a term. However the metatheory of λI_p variant is significantly more complicated than
 1054 the weak-head version. Since weak-head reduction was simpler and sufficient for our purposes (to
 1055 model object encodings) we opted for that variant. It would be interesting to study the full-cast
 1056 variant of λI_p with subtyping as well in future work.

1057 *Unified Algorithmic System.* In Section 5, we present an algorithmic version of λI_{\leq} . A notable
 1058 difference from the declarative system is that the typing and subtyping relation are defined sepa-
 1059 rately. An alternative design is to create an algorithmic unified subtyping relation directly from the
 1060 declarative version. The checking and synthesis judgments are denoted by $\Gamma \vdash e_1 \leq e_2 \Leftarrow A$ and
 1061 $\Gamma \vdash e_1 \leq e_2 \Rightarrow A$, respectively. However, the design is only a sketch and currently we do not have a
 1062 completeness proof for the unified algorithmic system.

1063 *Full Contravariance of Function Types.* As mentioned in Section 3.3, the unified subtyping rule of
 1064 function types is *partially* contravariant in the sense that bounds of function types are identical,
 1065 which follows the treatment of universal types in the Kernel Fun variant [Cardelli and Wegner 1985]
 1066 of System F_{\leq} . An alternative is to follow the *full* System F_{\leq} that allows bounds to be contravariant:

$$\begin{array}{c}
 \Gamma \vdash A' \leq A : \star \quad \Gamma \vdash e' \leq e : A \quad \Gamma, x \leq e' : A' \vdash B \leq B' : \star \\
 \Gamma \vdash e : A \quad \Gamma \vdash e' : A' \quad \Gamma \vdash A : \star \quad \Gamma, x \leq e : A \vdash B : \star \\
 \hline
 \Gamma \vdash (\Pi x \leq e : A. B) \leq (\Pi x \leq e' : A'. B') : \star \quad \text{S-PRODF}
 \end{array}$$

1067 We formulate an alternative system with such full contravariant rule and prove all lemmas in
 1068 Section 4 still hold. However, full System F_{\leq} is proved to be undecidable [Pierce 1992]. With
 1069 contravariance of bounds, λI_{\leq} using rule S-PRODF can subsume full System F_{\leq} , rendering the
 1070 system undecidable. Though we have not proved the decidability of λI_{\leq} yet, we adopt the Kernel-
 1071 Fun rule in λI_{\leq} and can at least rule out the undecidability caused by the full contravariance.

8 RELATED WORK

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 λC_{\leq} [Chen 1997], an extension of the calculus of constructions (λC) with subtyping. λC_{\leq} supports neither top types nor bounded quantification in order to simplify the metatheory. The proof of transitivity in λC_{\leq} is simpler and does not depend on strong normalization, though decidability still depends on strong normalization as in λC . Zwanenburg proposed PTS^{\leq} [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^{\leq} , 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^{\leq} .

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 λI_{\leq} shares the similar idea of being based on the subtyping relation, it has two major differences from PSS. First, λI_{\leq} 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, λI_{\leq} 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_{\leq} with contravariant arrow types, including the Kernel Fun variant [Cardelli and Wegner 1985].

Stratified Syntax with High-Order Subtyping. System F_{\leq}^{ω} 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_{\leq}^{ω} [Compagnoni 1995; Pierce and Steffen 1997] 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_{\leq}^{ω} . 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_{\leq}^{ω} contains a kinded subtyping judgment $\Gamma \vdash A \leq B : K$ which has a similar shape to the unified subtyping relation in λI_{\leq} . But the typing relation is separately defined in their system and not subsumed by the kinded subtyping judgment.

Stratified Subtyping Systems with Dependent Types. System λP_{\leq} [Aspinall and Compagnoni 1996] is a stratified system with dependent types and higher-order subtyping. The metatheory becomes more complex than System F_{\leq}^{ω} 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 λP_{\leq} does not support polymorphism (i.e. abstraction over types), bounded quantification or top types. System $\lambda \Pi_{\leq}$ [Castagna and Chen 2001; Chen 1998] is an improvement of λP_{\leq} . It has the property of type-level transitivity elimination, while System λP_{\leq}

has transitivity elimination only for normalized types. However, $\lambda\Pi_{\leq}$ is proved to be equivalent to λP_{\leq} in typing and subtyping, meaning that it has no increased expressiveness.

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] (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. However, the metatheory of DOL is not fully developed yet.

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 λI_{\leq} , 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. 2016, 2012, 2014; Rompf and Amin 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 λI_{\leq} , DOT subsumes System F_{\leq} 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 λI_{\leq} 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 λI_{\leq} .

Dependent Types with Explicit Casts. One key difference of λI_{\leq} to other systems with higher-order subtyping is the absence of a conversion rule. Instead, explicit casts are used for performing type-level computation. The motivation of using casts in λI_{\leq} is to decouple strong normalization from the proofs of metatheory, which also makes it possible to allow general recursion. There have been several studies [Kimmell et al. 2012; Sjöberg et al. 2012; Sjöberg and Weirich 2015; Stump et al. 2008; van Doorn et al. 2013; Weirich et al. 2013; Yang et al. 2016] working on using explicit casts instead of conversion rule in a dependently typed system. We follow the *iso-types* approach from λI calculus [Yang et al. 2016] which is a generalization of iso-recursive types. However, λI and other mentioned studies do not deal with subtyping in their systems.

9 CONCLUSION AND FUTURE WORK

This paper presented the λI_{\leq} calculus, a dependently typed calculus with unified syntax, which supports higher-order polymorphism, bounded quantification and top types. The calculus unifies typing and subtyping into a single relation, eliminating the circularity of typing and subtyping. The transitivity and type safety of λI_{\leq} are proved. For the future, we intend to show the decidability of the calculus. We already have a sound and complete algorithmic system and we believe it is decidable, though we do not have the proof yet. We also hope to explore several alternative designs of the calculus, such as full subtyping of recursive types, casts with full reduction and unified algorithmic subtyping. We would also like to incorporate the more general kinds of bounds of DOT into λI_{\leq} , and study whether that would suffice for λI_{\leq} to express common Scala programming idioms.

1177 **ACKNOWLEDGMENTS**

1178 We thank the anonymous reviewers for their helpful comments. We use the Ott tool [Sewell et al.
1179 2010] for typesetting and formalization of this paper. This work has been sponsored by the Hong
1180 Kong Research Grant Council Early Career Scheme project number 27200514.

1182 **REFERENCES**

- 1184 Andreas Abel and Dulma Rodriguez. 2008. Syntactic metatheory of higher-order subtyping. In *CSL '08*. 446–460.
- 1185 Beniamino Accattoli and Giulio Guerrieri. 2016. Open Call-by-Value. In *APLAS '16*. Springer, 206–226.
- 1186 Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. 2010. ΠΣ: Dependent types without the sugar. In *Functional and Logic Programming*. Springer, 40–55.
- 1187 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.
- 1188 Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *FOOL '12*. ACM.
- 1189 Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *OOPSLA '14*. ACM, 233–249.
- 1190 David Aspinall and Adriana Compagnoni. 1996. Subtyping dependent types. In *LICS '96*. 86–97.
- 1191 Lennart Augustsson. 1998. Cayenne — a Language with Dependent Types. In *ICFP '98*. ACM, 239–250.
- 1192 Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (1991), 125–154.
- 1193 Henk Barendregt. 1992. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, Vol. 2. 117–309.
- 1194 Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.
- 1195 Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. 1999. Comparing object encodings. *Information and Computation* 155, 1-2 (1999), 108–133.
- 1197 Joana Campos and Vasco T. Vasconcelos. 2015. Imperative objects with dependent types. In *FTfJP '15*. ACM, 2:1–2:6.
- 1198 Luca Cardelli. 1986a. Amber. In *Combinators and Functional Programming Languages*. Springer Berlin Heidelberg, 21–47.
- 1199 Luca Cardelli. 1986b. *A Polymorphic lambda-calculus with Type: Type*. Digital Systems Research Center.
- 1200 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.
- 1201 Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 471–523.
- 1202 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *POPL '14*. ACM, 33–45.
- 1203 Giuseppe Castagna and Gang Chen. 2001. Dependent types with subtyping and late-bound overloading. *Information and Computation* 168, 1 (2001), 1–67.
- 1204 Gang Chen. 1997. Subtyping calculus of construction. *Mathematical Foundations of Computer Science 1997* (1997), 189–198.
- 1205 Gang Chen. 1998. Dependent type system with subtyping (I) type level transitivity elimination. *Journal of Computer Science and Technology* 13, 6 (1998), 564–578.
- 1206 Gang Chen. 2003. Coercive Subtyping for the Calculus of Constructions. In *POPL '03*. ACM, 150–159.
- 1207 Adriana Compagnoni and Healfdene Goguen. 2003. Typed operational semantics for higher-order subtyping. *Information and Computation* 184, 2 (2003), 242–297.
- 1208 Adriana Beatriz Compagnoni. 1995. *Higher-order subtyping with intersection types*. Ph.D. Dissertation. University of Nijmegen.
- 1209 Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76 (1988), 95–120.
- 1210 Karl Cray, Robert Harper, and Sidd Puri. 1999. What is a recursive module?. In *PLDI '99*. ACM, 50–63.
- 1211 Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of subsumption, minimum typing and type-checking in $F \leq$. *Mathematical structures in computer science* 2, 01 (1992), 55–91.
- 1212 Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. Dissertation. Université Paris VII.
- 1213 Robert Harper. 2013. *Practical foundations for programming languages* (1st ed.). Cambridge University Press.
- 1214 DeLesley S. Hutchins. 2010. Pure Subtype Systems. In *POPL '10*. ACM, 287–298.
- 1215 Jonas Kaiser, Tobias Tebbi, and Gert Smolka. 2017. Equivalence of System F and $\lambda 2$ in Coq Based on Context Morphism Lemmas. In *CPP '17*. ACM, 222–234.
- 1216 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 *PLPV '12*. 15–26.
- 1217 Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Generics of a Higher Kind. In *OOPSLA '08*. ACM.

1225

- 1226 Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers
1227 University of Technology.
- 1228 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov,
1229 Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An Overview of the Scala Programming Language*. Technical
1230 Report IC/2004/64. EPFL Lausanne, Switzerland.
- 1231 Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. 2003. A nominal theory of objects with dependent
1232 types. In *ECOOP '03*. Springer, 201–224.
- 1233 Luca Paolini and Simona Ronchi Della Rocca. 1999. Call-by-value Solvability. *RAIRO-Theoretical Informatics and Applications*
1234 33, 6 (1999), 507–534.
- 1235 Benjamin C. Pierce. 1992. Bounded quantification is undecidable. In *POPL '92*. ACM, 305–315.
- 1236 Benjamin C. Pierce. 2002. *Types and programming languages*. MIT press.
- 1237 Benjamin C. Pierce and Martin Steffen. 1997. Higher-order subtyping. *Theoretical computer science* 176, 1 (1997), 235–282.
- 1238 Benjamin C. Pierce and David N. Turner. 1994. Simple type-theoretic foundations for object-oriented programming. *Journal*
1239 *of functional programming* 4, 02 (1994), 207–247.
- 1240 Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and*
1241 *Systems (TOPLAS)* 22, 1 (2000), 1–44.
- 1242 Tiark Rumpf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *OOPSLA '16*. ACM, 624–641.
- 1243 David A. Schmidt. 1994. *The Structure of Typed Programming Languages*. MIT Press.
- 1244 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, et al. 2010. Ott: Effective
1245 tool support for the working semanticist. *Journal of functional programming* 20, 1 (2010), 71–122.
- 1246 Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard,
1247 Aaron Stump, and Stephanie Weirich. 2012. Irrelevance, Heterogenous Equality, and Call-by-value Dependent Type
1248 Systems. In *MSFP '12*. 112–162.
- 1249 Vilhelm Sjöberg and Stephanie Weirich. 2015. Programming Up to Congruence. In *POPL '15*. ACM, 369–382.
- 1250 Martin Steffen. 1998. *Polarized Higher-Order Subtyping*. Ph.D. Dissertation. Technische Fakultät, Friedrich-Alexander-Uni-
1251 versität Erlangen-Nürnberg.
- 1252 Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. 2008. Verified Programming in Guru. In
1253 *PLPV '09*. 49–58.
- 1254 The Coq development team. 2016. *The Coq proof assistant reference manual*. <https://coq.inria.fr/refman/> Version 8.6.
- 1255 Floris van Doorn, Herman Geuvers, and Freek Wiedijk. 2013. Explicit Convertibility Proofs in Pure Type Systems. In *LFMTP*
1256 *'13*. ACM, 25–36.
- 1257 Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *ICFP '13*. ACM,
1258 275–286.
- 1259 Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* 115,
1260 1 (1994), 38–94.
- 1261 Hongwei Xi and Frank Pfenning. 1999. Dependent types in practical programming. In *POPL '99*. ACM, 214–227.
- 1262 Yanpeng Yang, Xuan Bi, and Bruno C. d. S. Oliveira. 2016. Unified Syntax with Iso-types. In *APLAS '16*. Springer, 251–270.
- 1263 Yanpeng Yang and Bruno C. d. S. Oliveira. 2017. Unifying Typing and Subtyping (Extended version). Available from
1264 <https://bitbucket.org/ypyang/oopsla17>. (2017).
- 1265 Jan Zwanenburg. 1999. Pure type systems with subtyping. In *TLCA '99*. 381–396.

1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274