Pure Subtype Systems

DeLesley S. Hutchins
MZA Associates Corporation
dhutchins@mza.com

Abstract

This paper introduces a new approach to type theory called *pure subtype systems*. Pure subtype systems differ from traditional approaches to type theory (such as pure type systems) because the theory is based on subtyping, rather than typing. Proper types and typing are completely absent from the theory; the subtype relation is defined directly over objects. The traditional typing relation is shown to be a special case of subtyping, so the loss of types comes without any loss of generality.

Pure subtype systems provide a uniform framework which seamlessly integrates subtyping with dependent and singleton types. The framework was designed as a theoretical foundation for several problems of practical interest, including mixin modules, virtual classes, and feature-oriented programming.

The cost of using pure subtype systems is the complexity of the meta-theory. We formulate the subtype relation as an abstract reduction system, and show that the theory is sound if the underlying reductions commute. We are able to show that the reductions commute locally, but have thus far been unable to show that they commute globally. Although the proof is incomplete, it is "close enough" to rule out obvious counter-examples. We present it as an open problem in type theory.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Languages, Theory

Keywords subtyping, dependent types, singleton types, transitivity elimination, abstract reduction systems

1. Introduction

Type theory has traditionally drawn a sharp distinction between *objects*, such as the number 3, and *proper types*, such as Int. Objects can be used in computations, such as $3 + 5 \longrightarrow 8$, whereas proper types cannot; $\mathsf{Int} + 5$ is not a valid expression. Types, on the other hand, can be used to quantify over terms, whereas objects cannot; the function $\lambda x : 3.\ x$ is not a valid expression either.

Because traditional type theory distinguishes between types and objects, it must also distinguish between typing and subtyping. Typing is relationship between objects and types, e.g. $3 : \mathsf{Int}$, whereas subtyping is a relationship between types, e.g. $\mathsf{Int} \leq \mathsf{Top}$.

POPL'10, January 17–23, 2010, Madrid, Spain.

This paper introduces a new approach to type theory that we call "pure subtype systems" which eliminates these distinctions. Pure subtype systems differ from traditional type theory in the following three ways:

1. Pure subtype systems do not distinguish between types and objects. Every term can behave as either a type or an object depending on context.
2. There is no typing relation. Typing is shown to be a special case of subtyping.
3. The subtype relation is defined over all terms, not just types.

In a pure subtype system, objects can act as types. The number 3 is interpreted as a *singleton type* – it denotes the set of all terms that are equal to 3. Likewise, the type Int can be used in computations: $\mathsf{Int} + 5 \longrightarrow \mathsf{Int}$. Performing a computation with types is very similar to abstract interpretation. A type denotes a set of possible values, so the computation returns a set of possible results – i.e. another type.

By eliminating the distinction between types and objects, we also eliminate the distinction between typing and subtyping. If objects are treated as singleton types, then the subtype relation can be extended to cover objects as well as types. Once this has been done, typing becomes superfluous, and can be eliminated from the theory. An alternative title for this paper that we considered is: "*Sub*-typing, a *sub*-stitute for typing."

1.1 Terminology

Before going on, it is important to clarify some terminology. The *terms* of a language include everything defined by the formal syntax of the language. Most type theories divide terms into two or three *sorts*, which are syntactically distinct. The two main sorts are *objects* and *types*; many theories also define *kinds*.

The word "object" here refers to the objects of the theory, not to "objects" in the sense of object-oriented programming. The word "term" refers to any valid piece of syntax. Some authors in the literature use the word "term" to denote objects, as distinct from types; our use of the word "term" includes both objects and types.

1.2 History and Motivation

Pure subtype systems were invented as part of a broader effort to develop a type theory for first-class, extensible, recursive modules [18] [19]. This work is described in the author's PhD thesis [19], which gives a theory of mixin modules and virtual classes based on pure subtype systems.

Modules present several challenges that are difficult to address with more traditional approaches. First, we wished to combine subtyping with dependent types, a combination that is tricky to work with [3]. Second, we found ourselves defining the subtype relation over objects as well as types.

Our theory of modules required dependent types because we defined modules as records (i.e. objects) which could contain type members. If $m.T$ denotes the type member T within the module m,

then $m.T$ is a dependent type — a type expression which depends upon the object m.

We used subtyping to handle module extension, which is similar to object-oriented inheritance. In an OO language, inheritance is intuitively related to subtyping because inheritance is a relationship between classes, and classes are types. However, module extension is a relationship between modules, and modules in our theory were objects, not types. Thus, we extended the subtype relation to include objects. Interestingly enough, Dreyer has come to a similar conclusion in his most recent work; MixML unifies signatures (i.e. types) and structures (i.e. modules) [14].

After working with the module system for a while, we realized that we had uncovered a new way of doing type theory in general, which was independent of the particular application to modules. This paper presents the basic ideas of pure subtype systems within the context of the familiar λ-calculus. We have based our presentation on Barendregt's successful theory of *pure type systems* [4].

1.3 Summary of key results

In this paper, we compare and contrast pure subtype systems (PSSs) with traditional approaches to type theory (Section 2), and we give some examples which illustrate how PSSs can be used (Section 3). We demonstrate that subtyping completely subsumes typing by embedding a pure type system in a PSS (Section 4), and we show that PSSs are type-safe if a property known as *transitivity elimination* holds (Section 5).

Subtyping is inherently harder to work with than typing, because subtyping is transitive, whereas typing is not. (Transitivity states that if $a \leq b$, and $b \leq c$, then $a \leq c$). The presence of a transitivity rule makes it hard to prove an inversion or generation lemma, which is a key part of most type safety proofs [26]. Standard practice is to prove a property called *transitivity elimination*, which shows that transitivity, in its general form, can be removed as an axiom of the system [26] [27] [10] [11].

In other theories of higher-order subtyping, such as System $F^{\omega}_{\leq}$, transitivity elimination is one of the most difficult results to prove (Section 8). Existing proofs rely on the fact that the language of types is strongly normalizing (i.e. the evaluation of every type expression is guaranteed to terminate). Pure subtype systems, like pure type systems, do not guarantee strong normalization, so existing proof techniques are not applicable.

Since a proof based on strong normalization is not possible, we have pursued a proof technique based on confluence instead (Section 6). We formulate the subtyping judgement as an abstract reduction system, and show that transitivity follows from commutativity of the underlying reductions. We are able to show that the reductions commute locally, but we have not been able to show that they commute globally.

Our partial proof is "close enough" to a full proof that the obvious counter-examples can be ruled out. However, the ultimate soundness of our technique remains an open question. Our hope is that by presenting this work as an open problem (Section 7), we can involve the larger research community in the quest for its solution.

2. From System $F^{\omega}_{\leq}$ to Pure Subtype Systems

We introduce pure subtype systems by way of an analogy. Pure subtype systems (or PSSs) generalize System $F^{\omega}_{\leq}$, the theory of higher-order subtyping, in much the same way that pure type systems (or PTSs) generalize System F_{ω}, the theory of higher-order types.

2.1 Higher-order types and PTSs

Barendregt's λ-cube classifies type theories into a hierarchy. The simply-typed λ-calculus includes only one form of abstraction: functions from objects to objects. System F adds polymorphism in the form of functions from types to objects, and System F_{ω} adds *higher-order types*, which are functions such as List that map from types to types.

System F_{ω} is considerably more complex than its predecessor, System F, because once functions have been added to the level of types, it is possible to construct and evaluate expressions at the level of types. In order to prove that such expressions are valid, type expressions are assigned *kinds*, in exactly the same way that object expressions are assigned types.

In System F_{ω} as it is usually presented [26], objects, types, and kinds are all syntactically distinct. System F_{ω} defines no less than six different pieces of syntax for λ-abstractions and their proper types and kinds. In the following table, t and u range over objects, T, U range over types, and K_1, K_2 range over kinds; x is an object variable, while X is a type variable:

description	abstraction	proper type or kind
ordinary functions	$\lambda x : T.\ u$	$T \to U$
polymorphic objects	$\lambda X : K.\ u$	$\Pi X : K.U$
type operators	$\lambda X : K_1.\ U$	$K_1 \to K_2$

Barendregt's theory of pure type systems (PTSs) demonstrates that this duplication of syntax is unnecessary. PTSs exploit the fact that the reduction and inference rules are identical for both object expressions and type expressions, so there is no need to distinguish syntactically between the two. PTSs have a single syntax for λ-abstractions, and a single syntax for proper types and kinds. Objects, types, and kinds are all terms, and the typing relation is defined uniformly over terms. In the following (much simpler) table for PTSs, s, t, u are terms, and x is a term variable:

abstractions	$\lambda x : t.\ u$	$\Pi x : t.\ s$

2.2 A uniform syntax for functions with subtyping?

System $F^{\omega}_{\leq}$ extends System F_{ω} with support for subtyping, and the subtyping judgement must be defined over all of the various pieces of syntax. It is tempting to try and reduce this complexity by using the same trick that Barendregt used for PTSs, but in System $F^{\omega}_{\leq}$, there is a problem. The following table lists the three forms of abstraction in System $F^{\omega}_{\leq}$:

ordinary functions	$\lambda x : T.\ u$	$T \to U$
polymorphic objects	$\lambda X \leq T.\ u$	$\Pi X \leq T.U$
type operators	$\lambda X \leq T.\ U$	$\Pi X \leq T.K$

Notice that these three forms of abstraction quantify over a variable in two different ways, using two different relations: typing (e.g. $x : T$) and subtyping (e.g. $X \leq T$).

A similar problem exists in System F_{ω} between typing and kinding. In order to unify the three forms of abstraction, PTSs combine typing and kinding into a single relation which is defined over all terms. If we are to unify the three forms of abstraction in System $F^{\omega}_{\leq}$, along the same lines as PTSs, *then we must necessarily combine typing and subtyping into a single relation which is defined over all terms*. This observation is the key insight that underlies pure subtype systems.

2.3 Unifying subtyping and kinding

The idea of using a single relation for both typing and subtyping sounds radical, but it actually has a precedent in the literature. The language of kinds in System $F^{\omega}_{\leq}$ is defined as:

$$K ::= * \mid \Pi X \leq T.\ K$$

In System $F^{\omega}_{\leq}$, every kind is associated with a Top-type, the supertype of all types which have that particular kind. The Top-type of a kind K, written $\mathrm{top}(K)$, is:

$$\begin{array}{lcl} \text{top}(*) & = & \mathsf{Top} \\ \text{top}(\Pi X \leq T.\ K) & = & \lambda X \leq T.\ \text{top}(K) \end{array}$$

Thus, every abstraction of the form $\lambda X : K.\ u$, in System F_ω, can be replaced with an abstraction of the form $\lambda X \leq \text{top}(K).\ u$, in System $F^\omega_\leq$, without loss of generality. In other words, *subtyping completely subsumes kinding*.

What we shall show in this paper is that by extending the subtype relation to cover all terms, subtyping can also subsume typing. Every abstraction of the form $\lambda x : T.\ u$ can be replaced with an abstraction of the form $\lambda x \leq t.\ u$ for some t. Instead of pure type systems, which are based upon typing, we obtain the theory of pure subtype systems, which are based entirely upon subtyping.

2.4 Unifying abstractions and proper types

Barendregt's pure type systems still maintain a distinction between functions, written $\lambda x : t.\ u$, and proper types, written $\Pi x : t.\ u$. Functions differ from Π-types in that functions are eliminated via β-reduction (i.e. $(\lambda x : t.\ u)(s) \longrightarrow [x \mapsto s]u$), whereas Π-types are eliminated by the following typing rule:

$$\frac{\Gamma \vdash\ f : (\Pi x : t.\ u) \quad \Gamma \vdash s : t}{\Gamma \vdash\ f(s) : [x \mapsto s]u}$$

As it turns out, the distinction between functions and Π-types is yet another needless duplication of syntax. Recent work by Kamareddine [21] has shown that Π-types are, in fact, unnecessary. She presents an alternative system in which the type of a function is simply another function, subject to the following constraint: if f is a function, and $f : F$, then $f(a) : F(a)$ for any valid argument a. In Kamareddine's system, the rule for eliminating Π-types is subsumed by ordinary β-reduction.

As it turns out, a very similar simplification arises in pure subtype systems. In section 2.3, we saw that the top-type of a kind $\Pi X \leq T.\ K$ is a function: $\lambda X \leq T.\ \text{top}(K)$. The act of replacing kinded quantification (i.e. $X : K$) with bounded quantification (i.e. $X \leq \text{top}(K)$) has a consequence: it also eliminates all Π-types, replacing them with λ-abstractions.

In pure subtype systems, the six pieces of syntax found in System $F^\omega_\leq$ for functions and proper types are all unified into a single piece of syntax: $\lambda x \leq t.\ u$.

2.5 Deconstructing the typing judgement

In most traditional type theories, including pure type systems, the type of a term provides three distinct pieces of information:

1. The type of a term describes the *shape* of the value that will be produced when the term is evaluated.
2. Terms which can be assigned types are said to be *well-typed*. The evaluation of a well-typed term will not generate type errors at run-time.
3. The type or kind of a term describes the level or universe in which the term resides.

In pure subtype systems, these three pieces of information are split into separate judgements which are largely orthogonal:

1. Subtyping compares the shape of terms.
2. The evaluation of a *well-formed* term will not generate type errors.
3. An (optional) universe judgement determines the universe in which the term resides.

Subtyping and typing have a significant amount of overlap because they are both judgements about shape. For example, the judgement $t : U \to S$ means that when t is evaluated, the result will be a function that accepts arguments of type U. Similarly, $T \leq (\lambda X \leq U.\ S)$ means that when the type expression T is evaluated, it will produce a function which accepts arguments that are subtypes of U.

However, typing also provides a piece of information that subtyping does not – a guarantee that evaluating the term will not produce any type errors. Subtyping can be used as a substitute for typing with regard to shape, but not with regard to program correctness. Pure subtype systems rely on a well-formedness judgement to ensure program correctness. The usual approach to proving type safety – "well-typed terms don't go wrong" – becomes "well-formed terms don't go wrong."

The third piece of information, universes, is optional. Traditional type theories usually include at least two universes: the universe of objects, and the universe of types. When Barendregt formulated pure type systems, he showed that the basic theory is the same for any universe structure; the universe structure is a parameter of the theory. The same principle holds for pure subtype systems. For the sake of simplicity, this paper describes a theory with only a single universe, but we will briefly show that it is possible to add other universe structures as well.

3. System $\lambda_\lhd$

Figure 1 introduces System $\lambda_\lhd$. System $\lambda_\lhd$ is the simplest pure subtype system that we could devise; it is a typed λ-calculus with a single universe.

Functions are written: $\lambda x \leq t.\ u$. Both the type bound t and the function body u are ordinary terms. Functions use bounded quantification rather than typing; the above function can be applied to any term that is a subtype of t.

Top is a universal supertype; every well-formed term t is a subtype of Top. This interpretation of Top differs from System $F^\omega_\leq$. In System $F^\omega_\leq$, Top is only a supertype of the proper types (the types of objects), it is not a supertype of type operators. As will be discussed in Section 4.2, our interpretation of Top makes System $\lambda_\lhd$ fully impredicative, which has important consequences for decidability.

Unlike pure type systems, the dependent type $\Pi x : t.\ u$ is not a part of the syntax; this type is subsumed by ordinary functions. The arrow type $t \to u$ is not present either, for the same reason. We treat $t \to u$ as syntactic sugar for an ordinary function $\lambda x \leq t.\ u$, where x is chosen fresh and does not appear in u.

3.1 Subtyping

Figure 1 gives the declarative formulation of subtyping, (as distinct from algorithmic subtyping, which we will introduce later), so the rules are named (DS-*Name*). Our naming scheme for rules distinguishes between *congruence rules* like (DS-APP), which compare terms of similar shape (i.e. functions with functions, or applications with applications), and *reduction rules* like (DS-EAPP), which compare terms of different shape.

In order to make the presentation as compact as possible, we use $\lhd$ as a metavariable that ranges over both the subtype relation ($\leq$), and the type equivalence relation ($\equiv$). This particular presentation style is the reason why we call the calculus System $\lambda_\lhd$. Each rule that is given in terms of $\lhd$ thus defines two different rules. For example, the rule for application (DS-APP) actually denotes the following:

$$\frac{\Gamma \vdash\ t \lhd t', \quad u \equiv u'}{\Gamma \vdash\ t(u) \lhd t'(u')} \quad \text{means}$$

$$\frac{\Gamma \vdash\ t \leq t', \quad u \equiv u'}{\Gamma \vdash\ t(u) \leq t'(u')} \quad \text{and} \quad \frac{\Gamma \vdash\ t \equiv t', \quad u \equiv u'}{\Gamma \vdash\ t(u) \equiv t'(u')}$$

Syntax:

x, y, z	Variable		
s, t, u ::=	Terms	Γ ::=	Type Contexts
x	variable	$\emptyset$	empty context
Top	universal supertype	$\Gamma, x \leq t$	variable
$\lambda x \leq t.\ u$	function		
$t(u)$	application	$\lhd$ ::=	Type relations
		$\leq$	subtype
v, w ::=	Values	$\equiv$	type equivalence
Top	universal supertype		
$\lambda x \leq t.\ u$	function		

Reduction: $\boxed{t \longrightarrow t'}$

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \quad \text{(E-Cong)}$$

$$(\lambda x \leq t.\ u)(s) \longrightarrow [x \mapsto s]u \quad \text{(E-App)}$$

Context well-formedness: $\boxed{\Gamma \text{ wf}}$

$$\emptyset \text{ wf} \quad \text{(W-Gam1)}$$

$$\frac{\Gamma \text{ wf} \quad x \notin \text{dom}(\Gamma) \quad \Gamma \vdash t \text{ wf}}{\Gamma, x \leq t \text{ wf}} \quad \text{(W-Gam2)}$$

Well-formedness: $\boxed{\Gamma \vdash t \text{ wf}}$

$$\frac{\Gamma \text{ wf} \quad x \in \text{dom}(\Gamma)}{\Gamma \vdash x \text{ wf}} \quad \text{(W-Var)}$$

$$\frac{\Gamma \text{ wf}}{\Gamma \vdash \mathsf{Top} \text{ wf}} \quad \text{(W-Top)}$$

$$\frac{\Gamma, x \leq t \vdash u \text{ wf}}{\Gamma \vdash \lambda x \leq t.\ u \text{ wf}} \quad \text{(W-Fun)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} \lambda x \leq s.\ \mathsf{Top} \quad \Gamma \vdash u \leq_{\text{wf}} s}{\Gamma \vdash t(u) \text{ wf}} \quad \text{(W-App)}$$

Well-subtyping: $\boxed{\Gamma \vdash t \lhd_{\text{wf}} u}$

$$\frac{\Gamma \vdash t \text{ wf}, \quad u \text{ wf} \quad \Gamma \vdash t \lhd u}{\Gamma \vdash t \lhd_{\text{wf}} u} \quad \text{(W-Sub)}$$

Subtyping: $\boxed{\Gamma \vdash t \lhd u}$

$$\frac{\Gamma \vdash s \lhd t, \quad t \lhd u, \quad t \text{ wf}}{\Gamma \vdash s \lhd u} \quad \text{(DS-Trans)}$$

$$\frac{\Gamma \vdash u \equiv t}{\Gamma \vdash t \equiv u} \quad \text{(DS-Sym)} \qquad \frac{\Gamma \vdash t \equiv u}{\Gamma \vdash t \leq u} \quad \text{(DS-Eq)}$$

$$\Gamma \vdash x \equiv x \quad \text{(DS-Var)}$$

$$\Gamma \vdash \mathsf{Top} \equiv \mathsf{Top} \quad \text{(DS-Top)}$$

$$\frac{\Gamma \vdash t \equiv t' \quad \Gamma, x \leq t \vdash u \lhd u'}{\Gamma \vdash \lambda x \leq t.\ u \ \lhd\ \lambda x \leq t'.\ u'} \quad \text{(DS-Fun)}$$

$$\frac{\Gamma \vdash t \lhd t', \quad u \equiv u'}{\Gamma \vdash t(u) \lhd t'(u')} \quad \text{(DS-App)}$$

$$\Gamma \vdash (\lambda x \leq t.\ u)(s) \equiv [x \mapsto s]u \quad \text{(DS-EApp)}$$

$$\Gamma \vdash t \leq \mathsf{Top} \quad \text{(DS-ETop)}$$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash x \leq t} \quad \text{(DS-EVar)}$$

Notation:

- C is a context with a hole: $C ::= [] \mid C(t) \mid t(C) \mid \lambda x \leq C.\ t \mid \lambda x \leq t.\ C$.
- $C[t]$ denotes the context C with the term t substituted for the hole $[]$ in the context.
- $[x \mapsto t]u$ denotes the capture-avoiding substitution of the term t for the variable x within u.
- $\text{fv}(t)$ denotes the set of free variables in the term t.
- $\text{dom}(\Gamma)$ denotes the set of variables defined in Γ.
- $x \leq t \in \Gamma$ is true if the type context Γ contains $x \leq t$.
- Note that $\lhd$ is a meta-variable which ranges over $\leq$ and $\equiv$.
- For compactness, we adopt the following convention:
  A pair of judgements $\Gamma \vdash J_1$ and $\Gamma \vdash J_2$ which are both made within the same type context Γ are written as $\Gamma \vdash J_1,\ J_2$.

Figure 1. System $\lambda_{\lhd}$ — syntax, operational semantics, and declarative subtyping

3.2 Point-wise subtyping

The subtyping judgement for System $\lambda_{\triangleleft}$ has been lifted almost verbatim from System $F^{\omega}_{\leq}$. In fact, System $\lambda_{\triangleleft}$ is essentially a fragment of System $F^{\omega}_{\leq}$, a fragment that only contains type operators.

Simpler theories of subtyping, such as System $F_{\leq}$, only define the subtype relation over *proper types* (i.e. arrow or Π-types), which are the types of objects. Each proper type denotes a set of objects. The subtype relation between proper types thus corresponds to the subset relation between sets; a type T is a subtype of U if and only if every object of type T is also an object of type U.

System $F^{\omega}_{\leq}$ extends the subtype relation to include type operators (a.k.a. higher-order types). However, type operators do not denote sets of objects; they denote functions over types. Thus, the idea that "subtypes are subsets" is no longer sufficient. The primary insight behind System $F^{\omega}_{\leq}$ is that *it is possible to define a meaningful subtype relation directly over functions*.

System $F^{\omega}_{\leq}$ uses a pointwise subtyping rule: given two type operators F and G, $F \leq G$ if and only if $F(A) \leq G(A)$ for any valid argument A. Notice the similarity between this definition of pointwise subtyping, and the definition of pointwise *typing* found in Kamareddine's theory, which was described in Section 2.4.

System $\lambda_{\triangleleft}$ uses the same point-wise subtyping rule found in System $F^{\omega}_{\leq}$. However, every term in System $\lambda_{\triangleleft}$ can be interpreted as a type, so every function can be interpreted as a type operator.

3.3 Contravariance

Subtyping in System $\lambda_{\triangleleft}$ differs from that in System $F^{\omega}_{\leq}$ in one respect: the argument type of a function is invariant rather than contravariant in subtypes. That is to say, $(\lambda x \leq t.\ u) \leq (\lambda x \leq t'.\ u)$ only if $t \equiv t'$. Although contravariance is a potentially useful extension, it also results in several well-known pathologies [25]. The practical applications of System $\lambda_{\triangleleft}$ that we have explored thus far (i.e. modules) do not require contravariance, so we have not included it in the theory. A further discussion of this issue can be found in [19].

3.4 Well-formedness

The subtype relation is defined only over terms that are well-formed. For clarity of presentation, however, we have chosen to define the $\leq$ relation syntactically, omitting the well-formedness checks. The complete subtype relation is written as $t \leq_{\text{wf}} u$, (pronounced "t is a well-subtype of u"). Any derivation of $t \leq_{\text{wf}} u$ only compares well-formed subterms of t and u.

The well-formedness judgement makes the following type checks:

- It ensures that the program is well-scoped: every variable x must be defined within the typing context.
- In a function application $t(u)$, it ensures that t is a function, and that u is a subtype of the argument type of that function.

3.5 Example: $3 \leq \mathsf{Nat}$

To illustrate how System $\lambda_{\triangleleft}$ works, we show how Nat, the type of natural numbers, and the number 3, can both be encoded in the standard way using Church numerals [26]. We further show that 3 is a subtype of Nat.

$$\begin{array}{lcl} \mathsf{Nat} & = & \lambda x \leq \mathsf{Top}.\ (x \to x) \to x \to x \\ & = & \lambda x \leq \mathsf{Top}.\ \lambda f \leq (\lambda y \leq x.\ x).\ \lambda a \leq x.\ x \\ 3 & = & \lambda x \leq \mathsf{Top}.\ \lambda f \leq (\lambda y \leq x.\ x).\ \lambda a \leq x.\ f(f(f(a))) \end{array}$$

Recall that the standard arrow-type $t \to u$ is syntax sugar for $\lambda y \leq t.\ u$. To show that $3 \leq \mathsf{Nat}$, we show that:
$\Gamma,\ f \leq (\lambda y \leq x.\ x),\ a \leq x \vdash\ f(f(f(a))) \leq x$, as follows:

$$\begin{array}{lcll} \Gamma,\ f \leq (\lambda y \leq x.\ x),\ a \leq x \vdash & & & \\ \quad f(f(f(a))) & \leq & (\lambda y \leq x.\ x)(f(f(a))) & \text{by DS-EVar} \\ & \equiv & x & \text{by DS-EApp} \end{array}$$

The encoding shown above is not specific to System $\lambda_{\triangleleft}$. It would be perfectly valid in System $F^{\omega}_{\leq}$ as well, if 3 and Nat were encoded at the level of types, rather than the level of objects.

3.5.1 Singleton types and abstract interpretation

Although System $\lambda_{\triangleleft}$ does not formally distinguish between types and objects, there is still a semantic difference between the two. The number 3 is a *singleton type*; it has no subtypes other than itself. (A subtype of 3 could only be constructed by finding a subtype of f, and that's not possible because f is a variable which does not appear in the bounds of other variables.)

The number 3 can be interpreted as either a type or an object. As an object, the number 3 can be used in computations, such as being added or multiplied by other numbers. When used as a type (e.g. $\lambda x \leq 3.\ x + x$), it is the type of all natural numbers that are equal to 3.

Similarly, Nat can be interpreted as either a type or an object. As a type, it is the type of natural numbers. As an object, it can be used in computations just like any other number. Using the standard Church encodings for addition and multiplication, we get:

$$\begin{array}{lcll} \mathsf{Nat} + n & = & \mathsf{Nat} & \text{for any } n \\ \mathsf{Nat} * n & = & \mathsf{Nat} & \text{for any } n \end{array}$$

The evaluation of such expressions is somewhat similar to abstract interpretation. When an abstract type like Nat is used as an object, it represents an unknown value. It is possible to perform computations with such values, but the result will also be unknown – i.e. a type.

3.6 Adding Universes

The number 3 is a subtype of Nat because they both have a similar shape. They are both functions, and they both accept the same number and the same type of arguments. This definition is sufficient to build a static type system.

However, in practical programming, we may also wish to ensure that a given computation will return an actual number, like 0 or 5, rather than a type, like Nat. Subtyping does not make this distinction, but it easy to add a universe judgement which does.

There are many ways in which a universe judgement can be constructed, with varying degrees of sophistication; what follows is one of the simplest ways. Terms are divided into two universes: 0 is the universe of objects, and 1 is the universe of types. In order to distinguish between these universes, the syntax of System $\lambda_{\triangleleft}$ must be extended so that variables are tagged with their universe. Object variables are written as x^0 or y^0, while type variables are written as x^1 or y^1:

$$\begin{array}{ll} J, K & ::= 0 \mid 1 \\ s, t, u & ::= x^K \mid \mathsf{Top} \mid \lambda x^K \leq t.\ u \mid t(u) \end{array}$$

Once variables have been tagged with their universe, the judgement $t \in \mathcal{U}(K)$ determines whether a term t is an object or a type:

$$\begin{array}{lcll} x^K & \in & \mathcal{U}(K) & \\ \mathsf{Top} & \in & \mathcal{U}(1) & \\ \lambda x^J \leq t.\ u & \in & \mathcal{U}(K) & \text{if } u \in \mathcal{U}(K) \\ t(u) & \in & \mathcal{U}(K) & \text{if } t \in \mathcal{U}(K) \end{array}$$

Note that a function is in universe K if its body is in K, regardless of what universe its argument is in. This simple model allows a function in any universe to quantify over any other universe, and thus supports both parametric polymorphism (objects that depend on types) and dependent types (types that depend on objects).

The well-formedness rule for function application must also be modified to ensure that function arguments are in the correct universe:

$$\frac{\Gamma \vdash\ t \leq_{\text{wf}} (\lambda x^K \leq s.\ \mathsf{Top}), \quad u \leq_{\text{wf}} s \quad u \in \mathcal{U}(K)}{\Gamma \vdash\ t(u)\ \text{wf}}$$

It is easy to see that universes are preserved under β-reduction, because a variable in universe K is replaced with a term in universe K. Moreover, by extending our definitions of 3 and Nat with universe tags, it also clear that 3 is an object (i.e. $3 \in\ \mathcal{U}(0)$), and Nat is a type:

$$\mathsf{Nat}{=}\lambda x^1 \leq \mathsf{Top}.\ \lambda f^0 \leq (x^1 \to x^1).\ \lambda a^0 \leq x^1.\ x^1$$
$$3\ \ {=}\lambda x^1 \leq \mathsf{Top}.\ \lambda f^0 \leq (x^1 \to x^1).\ \lambda a^0 \leq x^1.\ f^0(f^0(f^0(a^0)))$$

We present universes as a curiosity. The universe judgement shown here is completely orthogonal to subtyping, so the presence or absence of universes does not affect any of the results that we present in this paper. The subtype relation is still defined over all terms in all universes. In particular, subtyping can cross universe boundaries: objects are still subtypes of types. In the interest of simplicity, the rest of this paper will be devoted to the version of System $\lambda_{\triangleleft}$ with only a single universe.

4. Embedding of a Pure Type System

To show that pure subtype systems are comparable in expressive power to pure type systems, we show that System $\lambda*$ can be embedded in System $\lambda_{\triangleleft}$. System $\lambda*$ is a PTS described by Barendregt [4], which supports polymorphism, type operators, and dependent types. It has a single sort $*$, and the typing relation $* : *$. The embedding is as follows:

$$\begin{array}{rcl} \langle x \rangle & = & x \\ \langle * \rangle & = & \mathsf{Top} \\ \langle \lambda x : t.\ u \rangle & = & \lambda x \leq \langle t \rangle.\ \langle u \rangle \\ \langle \Pi x : t.\ u \rangle & = & \lambda x \leq \langle t \rangle.\ \langle u \rangle \\ \langle t(u) \rangle & = & \langle t \rangle (\langle u \rangle) \\ \\ \langle \emptyset \rangle & = & \emptyset \\ \langle \Gamma, x : t \rangle & = & \langle \Gamma \rangle, x \leq \langle t \rangle \end{array}$$

Lemma 4.1 (Substitution is preserved under translation)
$\langle [x \mapsto t]u \rangle = [x \mapsto \langle t \rangle]\langle u \rangle$

Proof: By straightforward induction on u. □

Theorem 4.2 (Reduction is preserved under translation)
If $t \longrightarrow u$ then $\langle t \rangle \longrightarrow \langle u \rangle$.

Proof: By induction on the derivation of $t \longrightarrow u$, using lemma 4.1 for the base case. □

Theorem 4.3 (Typing is preserved under translation)
If $\Gamma \vdash t : u$ then $\langle \Gamma \rangle \vdash \langle t \rangle \leq_{\text{wf}} \langle u \rangle$.

Proof: By induction on the derivation of $t : u$. Full details can be found in the author's PhD thesis [19]. □

4.1 Caveat: Pure Type Systems and Universes

The calculus of constructions can be embedded in System $\lambda_{\triangleleft}$ using a similar technique, as can all the other members of Barendregt's λ-cube. The embedding shown above thus demonstrates that subtyping is expressive. However, our embedding does not enforce the restrictions that differentiate the various members of the λ-cube.

Every PTS comes equipped with a universe structure (i.e. the *sorts*) and a set of rules that restricts the ways in which functions in one universe can quantify over other universes. It is easy enough to add universes to a pure subtype system, but it is not so easy to enforce universe restrictions. Pure subtype systems have an additional symmetry because they unify λ-abstractions and Π-types, and that symmetry means that anything which is allowed in one universe must be allowed in the others.

For example, the simply-typed λ-calculus has two universes, objects and types, but it only permits functions that map from objects to objects. However, in a pure subtype system the arrow-types become functions, e.g. $\mathsf{Int} \to \mathsf{Int}$ becomes $\lambda x \leq \mathsf{Int}.\ \mathsf{Int}$. There is thus no way to permit functions from objects to objects without also permitting functions from objects to types. There is no "simply-typed" version of a pure subtype system.

4.2 Impredicativity and Girard's Paradox

Although System $\lambda*$ is both elegant and expressive, it also has a well-known flaw. It admits Girard's paradox, and is thus not strongly normalizing [4]. Its embedding thus demonstrates that System $\lambda_{\triangleleft}$ is not strongly normalizing either.

Theorem 4.4 (System $\lambda_{\triangleleft}$ is not strongly normalizing.)

Proof: According to Girard's paradox, there exists a well-typed term in System $\lambda*$ that has no normal form, and thus has an infinite reduction sequence [4]. According to theorem 4.3, the translation of this term is well-formed in System $\lambda_{\triangleleft}$, and by theorem 4.2, the translation also has an infinite reduction sequence. □

A skeptic might argue that Girard's paradox in System $\lambda*$ stems from the circular $* : *$ rule, which confuses the universe of types and the universe of objects, and since System $\lambda_{\triangleleft}$ confuses the two universes much more thoroughly, it is not surprising that it suffers from the same problem. However, this argument is not correct. As Barendregt explains, Girard's paradox can also be found in System λU, another pure type system which has no such circularity [4].

The cause of Girard's paradox is impredicativity. A Π-type is *impredicative* if the type variable quantifies over the Π-type itself. Such types can be used to build functions that can be applied to themselves, which are the basis for non-terminating expressions.

It is possible for a type theory to be both impredicative and strongly normalizing; Girard's System F [16] and Luo's extended calculus of constructions (ECC) [23] are two well-known examples. However, great care is required. Both of these systems place tight controls on impredicativity; System F controls it by being relatively simple, while ECC uses a sophisticated universe structure that restricts impredicativity to the universe of propositions; all other universes are predicative.

System $\lambda_{\triangleleft}$ suffers from Girard's paradox because it places no constraints on impredicativity. All terms belong to a single universe, and Top is both a member of the universe, and a type that ranges over all elements of the universe.

We believe, but have not yet proven, that strong normalization could be restored to pure subtype systems by either removing Top, which is the source of the impredicativity, or by using a more sophisticated universe judgement with stronger restrictions. In the meantime, however, we shall explore the meta-theory of pure subtype systems under the assumption that strong normalization does not hold, just as Barendregt does when developing the meta-theory for pure type systems.

5. Type Safety

This section shows that System $\lambda_{\triangleleft}$ is type-safe so long as subtyping has the transitivity elimination property. Transitivity elimination is the subject of the Section 6.

Our proof of type safety is an adaptation of the standard technique of progress and preservation [31]. In a traditional type system, progress states that "well-typed terms don't get stuck"; i.e. if $t : T$, then t must either be a value, or there exists a t' such that $t \longrightarrow t'$. Preservation states that reducing a term will not change its type; if $t : T$ and $t \longrightarrow t'$, then $t' : T$.

In System $\lambda_{\lhd}$ we prove a similar result for subtyping and well-formedness. We show that "well-formed terms don't get stuck", and well-formedness is preserved under reduction. The following proof is only a sketch; the complete proof can be found in [19]. Note that we use a, b, c in addition to s, t, u as meta-variables for terms.

Conjecture 5.1 (Transitivity elimination)
If $\Gamma \vdash\ v \leq_{\text{wf}} w$, then there exists a proof of $\Gamma \vdash\ v \leq w$ that ends in either (DS-FUN) or (DS-ETOP).

Lemma 5.2 (Inversion of subtyping — declarative version)
If $\Gamma \vdash\ (\lambda x \leq t.\ u) \leq_{\text{wf}} (\lambda x \leq t'.\ u')$ then $\Gamma \vdash\ t \equiv t'$.

Proof: By conjecture 5.1 (transitivity elimination). □

Lemma 5.3 (Reduction implies equivalence)
If $t \longrightarrow t'$, then $\Gamma \vdash\ t \equiv t'$.

Proof: By induction on the derivation of $t \longrightarrow t'$. The base case is by rule (DS-EAPP). □

Lemma 5.4 (Substitution)
If $\Gamma, x \leq t, \Gamma' \vdash\ u \lhd_{\text{wf}} s$ and $\Gamma \vdash\ t' \leq_{\text{wf}} t$
then $\Gamma, [x \mapsto t']\Gamma' \vdash\ [x \mapsto t']u \lhd_{\text{wf}} [x \mapsto t']s$.

Proof: By induction on the derivation of $u \lhd_{\text{wf}} s$. Every derivation of the form x wf is replaced with t' wf, and every derivation of the form $x \leq t$ is replaced with $t' \leq t$. □

Theorem 5.5 (Progress)
If $\emptyset \vdash t$ wf then either $t = v$ for some v (i.e. t is a value), or there exists a t' such that $t \longrightarrow t'$.

Proof: By induction on the derivation of t wf. The proof is by straightforward analysis of cases; see [19] for details. □

Theorem 5.6 (Preservation)
If $\Gamma \vdash t \leq_{\text{wf}} u$ and $t \longrightarrow t'$ then $\Gamma \vdash t' \leq_{\text{wf}} u$.

Proof: By induction on the derivation of t wf.

The proof has two parts.

For the first part of the proof, we show that if $t \leq u$, and $t \longrightarrow t'$, then $t' \leq u$. By lemma 5.3 (reduction implies equivalence) $t \equiv t'$. It follows that $t' \leq u$ using rule (DS-TRANS).

For the second part of the proof, we show that t wf and $t \longrightarrow t'$ implies t' wf. The proof is by induction on the derivation of t wf, and the most interesting case is as follows:

Case $t = (\lambda x \leq a.\ b)(c) \longrightarrow [x \mapsto c]b$.
The two premises of t wf are $(\lambda x \leq a.\ b) \leq_{\text{wf}} (\lambda x \leq a'.\ \mathsf{Top})$, and $c \leq_{\text{wf}} a'$. We have $a \equiv a'$ by lemma 5.2 (inversion of subtyping), which gives us $c \leq_{\text{wf}} a$ by rule (DS-TRANS). We then have $[x \mapsto c]b$ wf by lemma 5.4 (substitution). □

6. Transitivity Elimination

The definition of subtyping given in Figure 1 is known as *declarative subtyping*. The declarative definition is easy to read, and it is also easy to prove certain lemmas, such as substitution and narrowing. However, the declarative definition is problematic because it includes a transitivity rule:

$$\frac{\Gamma \vdash\ s \leq t, \quad t \leq u, \quad t \text{ wf}}{\Gamma \vdash\ s \leq u} \text{ (DS-TRANS)}$$

Transitivity is troublesome for two reasons. The first problem is that declarative subtyping is not an algorithm, because it is not syntax-directed; there is a term t in the premises that is absent in the conclusion. There is therefore a practical need to find a different formulation of subtyping that can be implemented within a compiler.

The second, more serious problem is that the presence of a transitivity rule prevents us from completing the proof of type safety. The type safety proof given earlier has the following step: given a well-formed redex $(\lambda x \leq a.\ b)(c)$, we must show that $[x \mapsto c]b$ is well-formed. This seems like a straightforward application of the substitution lemma, but the substitution lemma requires that $c \leq a$. We do not actually have a proof that $c \leq a$; instead, well-formedness tells us that:

(1) $(\lambda x \leq a.\ b) \leq (\lambda x \leq a'.\ \mathsf{Top})$ and
(2) $c \leq a'$

If the derivation of (1) ends in rule (DS-FUN), then we have $a \equiv a'$, and consequently $c \leq a$, so the substitution lemma applies. However, if the derivation of (1) ends in rule (DS-TRANS), then there is no immediate relationship between a and a'. If a and a' are unrelated, then the substitution lemma cannot be applied, and the proof of type safety cannot be completed.

The standard technique for resolving this problem is to reformulate the subtype relation into an algorithmic form that does not include a transitivity rule, a process called *transitivity elimination* [26] [27] [10] [11]. In essence, transitivity elimination is a proof that subtyping is sound. After all, if the subtyping rules allowed us to derive that $(\lambda x \leq a.\ b) \leq (\lambda x \leq a'.\ \mathsf{Top})$, where $a \not\equiv a'$, then there would clearly be an error in the theory.

The remainder of this section provides a partial proof of transitivity elimination. Although the proof is incomplete, our proof technique is novel, and we believe it offers insight into the fundamental problem.

6.1 Subtyping as an abstract reduction system

The problems caused by the transitivity rule are not restricted to subtyping. They arise in any system which has a non-trivial notion of type equivalence, including all higher-order type theories. Equivalence is a relation which is reflexive, symmetric, and transitive. The standard technique for dealing with equivalence is to formulate the equivalence rules as an *abstract reduction system* (ARS) [26]. In an ARS, $t \equiv u$ if and only if there exists an s such that $t \twoheadrightarrow s \twoheadleftarrow u$.

ARSs do not have a symmetry or transitivity rule, so transitivity elimination is an immediate property of the system. Moreover, if the reduction system is *confluent*, then transitivity is *admissible*, which means that it can be derived from first principles.

We adopt this same technique for algorithmic subtyping in System $\lambda_{\lhd}$. Unlike equivalence, subtyping is an inequality rather than an equality, and that affects the way in which we define the relation.

The algorithmic formulation of subtyping for System $\lambda_{\lhd}$ is presented in Figure 2. Most of the rules in the declarative system, including (DS-VAR), (DS-TOP), (DS-APP) and (DS-FUN), involve comparisons between terms which have the same shape. These rules become congruence rules in the reduction system.

That leaves three remaining rules to consider: (DS-EAPP), (DS-EVAR), and (DS-ETOP). We reformulate these three rules as reduction rules. An equivalence reduction, written $t \xrightarrow{\equiv} t'$, de-

Prevalidity: $\boxed{\Gamma \text{ prevalid}}$

$$\emptyset \text{ prevalid} \quad \text{(P-CTX1)}$$

$$\frac{\Gamma \text{ prevalid} \quad x \notin \text{dom}(\Gamma) \quad \text{fv}(t) \subseteq \text{dom}(\Gamma)}{\Gamma, x \leq t \text{ prevalid}} \quad \text{(P-CTX2)}$$

Subtyping: $\boxed{\Gamma \vdash_A t \lhd u}$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A t \lhd t} \quad \text{(AS-REFL)}$$

$$\frac{\Gamma \vdash_A t \xrightarrow{\lhd} t', \quad t' \lhd u}{\Gamma \vdash_A t \lhd u} \quad \text{(AS-LEFT)}$$

$$\frac{\Gamma \vdash_A u \xrightarrow{\equiv} u', \quad t \lhd u'}{\Gamma \vdash_A t \lhd u} \quad \text{(AS-RIGHT)}$$

Transitive Subtyping: $\boxed{\Gamma \vdash_A t \lhd^* u}$

$$\frac{\Gamma \vdash_A s \lhd u}{\Gamma \vdash_A s \lhd^* u} \quad \text{(AST-SUB)}$$

$$\frac{\Gamma \vdash_A s \lhd^* t, \quad t \lhd^* u}{\Gamma \vdash_A s \lhd^* u} \quad \text{(AST-TRANS)}$$

Subtype reduction: $\boxed{\Gamma \vdash_A t \xrightarrow{\leq} t'}$

$$\frac{\Gamma \text{ prevalid} \quad x \leq t \in \Gamma}{\Gamma \vdash_A x \xrightarrow{\leq} t} \quad \text{(SRS-PROM)}$$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A t \xrightarrow{\leq} \mathsf{Top}} \quad \text{(SRS-TOP)}$$

Equivalence reduction: $\boxed{\Gamma \vdash_A t \xrightarrow{\equiv} t'}$

$$\frac{\Gamma \vdash_A s \leq^* t}{\Gamma \vdash_A (\lambda x \leq t.\, u)(s) \xrightarrow{\equiv} [x \mapsto s]u} \quad \text{(SRE-APP)}$$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A \mathsf{Top}(t) \xrightarrow{\equiv} \mathsf{Top}} \quad \text{(SRE-TOPAPP)}$$

Congruence rules: $\boxed{\Gamma \vdash_A t \xrightarrow{\lhd} t'}$

$$E_{\equiv} ::= [] \mid E_{\equiv}(t) \mid t(E_{\equiv}) \mid \lambda x \leq E_{\equiv}.\, t$$

$$E_{\leq} ::= [] \mid E_{\leq}(t)$$

$$\frac{\Gamma \vdash_A t \xrightarrow{\lhd} t'}{\Gamma \vdash_A E_{\lhd}[t] \xrightarrow{\lhd} E_{\lhd}[t']} \quad \text{(SR-CONG)}$$

$$\frac{\Gamma, x \leq t \vdash_A u \xrightarrow{\lhd} u'}{\Gamma \vdash_A \lambda x \leq t.\, u \xrightarrow{\lhd} \lambda x \leq t.\, u'} \quad \text{(SR-FUN)}$$

$$\frac{\Gamma \vdash_A t \xrightarrow{\equiv} t'}{\Gamma \vdash_A t \xrightarrow{\leq} t'} \quad \text{(SR-EQ)}$$

Figure 2. System $\lambda_{\lhd}$— algorithmic subtyping

.

notes a rewrite step which produces a term t' that is equivalent to t. β-reduction (SRE-APP) falls into this category. Subtype reduction, written $t \xrightarrow{\leq} t'$, denotes a rewrite step which produces a term t' that is a supertype of t. Variable promotion (SRS-PROM) and Top-promotion (SRS-TOP) fall into this category.

An equivalence reduction step $t \xrightarrow{\equiv} t'$ can be applied anywhere in a term. A subtype reduction step $t \xrightarrow{\leq} t'$ can only be applied in positive (i.e. covariant) positions within a term. Positive positions are limited to function bodies and the left-hand side of applications. We express this requirement by means of two evaluation contexts: $E_{\equiv}$ may have a hole in any position, whereas $E_{\leq}$ may only have a hole in positive positions. Neither evaluation context can step inside a λ-abstraction. Unlike ordinary reduction, both subtype and equivalence reduction must be done within a context Γ that assigns bounding types to variables. Rule (SR-FUN) is used to reduce terms inside λ abstractions.

Within this framework, we define type equivalence and subtyping as follows, where $\xrightarrow{\lhd}\!\!\!\!\rightarrow$ denotes the reflexive and transitive closure of $\xrightarrow{\lhd}$, and $u \xleftarrow{\equiv} t$ means the same thing as $t \xrightarrow{\equiv} u$.

- $\Gamma \vdash_A t \equiv u$ iff $\Gamma \vdash_A t \xrightarrow{\equiv}\!\!\!\!\rightarrow s \xleftarrow{\equiv} u$ for some s.
- $\Gamma \vdash_A t \leq u$ iff $\Gamma \vdash_A t \xrightarrow{\leq}\!\!\!\!\rightarrow s \xleftarrow{\equiv} u$ for some s.

The definition of type equivalence is standard. Two terms t and u are equivalent if they both reduce to a common term. The definition of subtyping is similar. The only difference is that for subtyping, the reduction sequence for t may promote variables to supertypes, or subterms to Top as necessary.

Within this framework, basic meta-theoretic properties of subtyping follow directly from standard properties of the corresponding reductions. In particular, if $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ commute, then subtyping is transitive. If $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ were strongly normalizing (which they are not), then the subtype judgement would be decidable.

6.2 Resolving circularities

In the declarative definition of subtyping, the subtype relation is defined only over well-formed terms. This introduces a circularity between subtyping and well-formedness that is difficult to unravel. The algorithmic definition of subtyping breaks this circularity by defining the subtype relation over all terms, not just well-formed ones, and that necessitates a few changes to the system.

The first change is there is a new reduction rule: (SRE-TOPAPP). Since algorithmic subtyping is defined over ill-formed terms, we must supply an interpretation for ill-formed applications. The need for this rule is explained in section 6.6.1.

The second change is that rule (SRE-APP) has a premise that does not exist in the declarative system. The redex $(\lambda x \leq t.\, u)(s)$ can only be eliminated if s is a subtype of t. This premise will

always be satisfied if the redex is well-formed. However, since redexes may not be well-formed, the algorithmic system performs a "dynamic type check" before eliminating the redex.

It is important to notice that the premise of (SRE-APP) is defined with $\leq^*$, the transitive closure of subtyping, rather than $\leq$, which is ordinary subtyping. There are two reasons for this decision. Both reasons are related to the fact that, in the proofs that follow, we shall attempt to show that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute.

The first reason is that the reduction rules constitute a *conditional rewrite system* [5], because rule (SRE-APP) has a premise, or condition. If the condition were specifed as $s \leq t$, then it would be a *join condition*, and Bergstra and Klop have shown that join conditions cause commutativity to fail [5]. By defining the condition with $\leq^*$, we avoid this pitfall.

The second reason is related. The diagrams of commutativity make use of two key lemmas: substitution and narrowing, in order to show that the premise of (SRE-APP) is satisfied on both sides of the diagram. Both of these lemmas require transitivity. Transitivity follows from commutativity, but that would be a circular proof. This potential circularity is eliminated by using $\leq^*$, which is transitive by definition.

6.3 Confluence and Commutativity

We shall now attempt to prove that transitivity is admissible in the algorithmic system. This property is the same as transitivity elimination in the declarative system; we must show that any subtype derivation which makes use of transitivity can be rewritten as one that that does not.

In a simple ARS, transitivity follows from confluence. However, algorithmic subtyping is not simple, because there are two kinds of reduction: $\xrightarrow{\leq}$, and $\xrightarrow{\equiv}$. Moreover, the two kinds of reduction are not orthogonal; $\xrightarrow{\equiv}$ implies $\xrightarrow{\leq}$ by (SR-EQ). The exact property that we require is thus more specific than ordinary confluence.

The $\xrightarrow{\leq}$ relation by itself is trivially confluent, because ($t \xrightarrow{\leq} \mathsf{Top}$) can be used as the completing edges of any diagram. The two non-trivial properties that we are interested in are shown below.

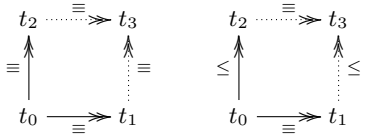
Theorem 6.1 ($\xrightarrow{\equiv}$ is confluent)

If $\Gamma \vdash_A\ t_0 \xrightarrow{\equiv}\!\!\!\!\!\rightarrow\ t_1,\ t_0 \xrightarrow{\equiv}\!\!\!\!\!\rightarrow\ t_2$, then there exists a t_3, such that $\Gamma \vdash_A\ t_1 \xrightarrow{\equiv}\!\!\!\!\!\rightarrow\ t_3,\ t_2 \xrightarrow{\equiv}\!\!\!\!\!\rightarrow\ t_3$.

Conjecture 6.2 ($\xrightarrow{\equiv}$ commutes with $\xrightarrow{\leq}$)

If $\Gamma \vdash_A\ t_0 \xrightarrow{\equiv}\!\!\!\!\!\rightarrow\ t_1,\ t_0 \xrightarrow{\leq}\!\!\!\!\!\rightarrow\ t_2$ then there exists a t_3, such that $\Gamma \vdash_A\ t_2 \xrightarrow{\equiv}\!\!\!\!\!\rightarrow\ t_3,\ t_1 \xrightarrow{\leq}\!\!\!\!\!\rightarrow\ t_3$.

These properties are illustrated by the following diagrams:

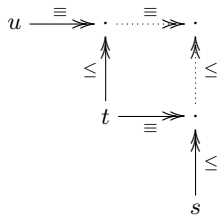


The solid lines of a confluence diagram are the *spanning edges*, and represent the premises, while the dotted lines are the *completing edges*, and represent the conclusion.

Lemma 6.3 (Commutativity implies transitivity)

If $\xrightarrow{\equiv}$ commutes with $\xrightarrow{\leq}$, then transitivity is admissible.

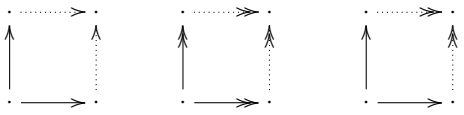
Proof: Assume we have $\Gamma \vdash_A\ s \lhd t$ and $\Gamma \vdash_A\ t \lhd u$. If $\xrightarrow{\equiv}$ commutes with $\xrightarrow{\leq}$, then we can construct a derivation of $\Gamma \vdash_A\ s \lhd u$, according to the following diagram:



□

6.4 Confluence: a brief digression

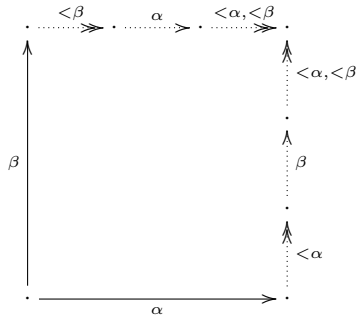
Confluence (or commutativity) is a fundamental property of any abstract reduction system. However, the idea of "confluence" encompasses three properties of interest:



It is seldom possible to prove global confluence directly, because the spanning edges of the diagram have an arbitrary number of reductions. Instead, most proofs start with either the diamond property or local confluence, which have only a single reduction on the spanning edges, and can thus be proven by simple analysis of all possible cases.

The diamond property implies gobal confluence because any global confluence diagram can be filled in, or "tiled" with single-step diagrams. Local confluence can also be used to "tile" a diagram, but in the case of local confluence, the tiling process is not guaranteed to terminate, because local confluence "tiles" may have multiple reductions on their completing edges.

There are several ways to prove global confluence from local confluence, of which Van Oostrom's technique of decreasing diagrams is perhaps the most general [30] [22]. The decreasing diagrams technique states that a reduction system is globally confluent if it is locally confluent, and if the elementary diagrams of local confluence have the following form:



The technique assigns indices (i.e. α, β) to reductions. The indices are drawn from a set with a well-founded order. Each of the additional reductions on the completing edges have a smaller index than the reductions on the spanning edges, and this fact is used to show that the tiling process terminates.

6.5 Proof that $\xrightarrow{\equiv}$ is confluent

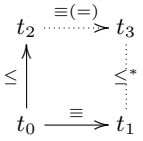
Our proof is an adaptation of Takahashi's proof of confluence for the untyped λ-calculus [29] [6], which shows that simultaneous reduction has the diamond property. Full details can be found in [19].

6.6 Commutativity of $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$

The last step in the proof of transitivity elimination is to show that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute. Unfortunately, it is here that our proof technique breaks down. We can show that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute locally, but we have thus far been unable to show that they commute globally. In fact, the actual property we prove is not even local commutativity per se, but something quite similar:

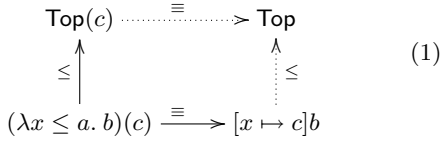
Lemma 6.4 ($\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute locally)
If $\Gamma \vdash_A\ t_0 \xrightarrow{\equiv} t_1,\ t_0 \xrightarrow{\leq} t_2$, then there exists a t_3, such that $\Gamma \vdash_A\ t_2 \xrightarrow{\equiv}{}^{(=)}\ t_3$, and $\Gamma \vdash_A\ t_1 \leq^* t_3$, where $\xrightarrow{\equiv}{}^{(=)}$ is the reflexive closure of $\xrightarrow{\equiv}$.

This property is illustrated by the following diagram. Note that the completing edge of the diagram on the right-hand side is not a subtype reduction (which would prove commutativity) but a transitive subtyping judgement, which we represent by drawing a dotted line instead of an arrow.

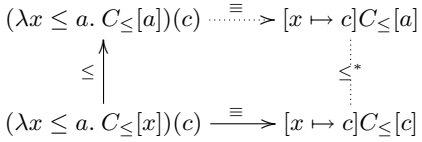


6.6.1 Case analysis

A full analysis of all cases can be found in [19]. The two main cases of interest are shown below.



(1)



(2)

Case (1) illustrates why (SRE-TopApp) is necessary. The term $\mathsf{Top}(c)$ is not well-formed, but we must handle it because algorithmic subtyping is defined over all terms, not just well-formed terms.

Case (2) is important because it is the only case which requires $\leq^*$ as the completing edge; every other case has a single subtype reduction on the completing edge. Were it not for case (2), $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ would have the diamond property, and would thus commute globally.

In case (2), one edge of the diagram promotes a variable x to its bounding type a. (The notation $C_{\leq}[x]$ represents a term in which x appears in a covariant position.) The other edge contracts the redex, replacing x with c. The premise of (SRE-App) tells us that $c \leq^* a$, and the congruence rules for reduction allow us to derive $C_{\leq}[c] \leq^* C_{\leq}[a]$ on the completing edge.

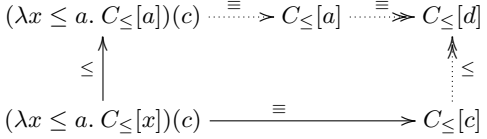
However, notice what has happened: a simple promotion $C_{\leq}[x] \xrightarrow{\leq} C_{\leq}[a]$ on the left-hand side becomes a full subtyping judgement $C_{\leq}[c] \leq^* C_{\leq}[a]$ on the right-hand side.

6.6.2 Commutativity by decreasing diagrams

Although the property described by Lemma 6.4 is not technically local commutativity, it can be used to "tile" global commutativity diagrams in much the same way, because the $\leq^*$ relation is defined in terms of reduction. Moreover, the property is compatible with Van Oostrom's technique of decreasing diagrams, which we show as follows.

Assume we have assigned indices to reductions, as required by Van Oostrom's technique. The proof of commutativity then proceeds by induction. For the base case, we show that commutativity holds for reduction sequences of index zero. We then show that if commutativity holds for reduction sequences of index at most n, then it holds for sequences of index at most $n + 1$.

In case (2) above, if the reductions in $c \leq^* a$ have indices which are strictly smaller than the spanning edges of the diagram, then we can use commutativity as an induction hypothesis. Applying lemma 6.3 (transitivity elimination) to $c \leq^* a$, yields $c \xrightarrow{\leq}\!\!\!\!\!\rightarrow\ d \xleftarrow{\equiv} a$, which gives us a diagram of true local confluence:



Van Oostrom's decreasing diagrams technique guarantees that indices never increase during the tiling process. The indices of the additional reductions on the completing edges (i.e. $C_{\leq}[c] \xrightarrow{\leq}\!\!\!\!\!\rightarrow C_{\leq}[d] \xleftarrow{\equiv} C_{\leq}[a]$) are therefore no greater than the indices of $c \leq^* a$, which are strictly smaller than the spanning edges. The above diagram of local confluence is thus a decreasing diagram, and the rest of Van Oostrom's technique can be applied as usual.

To summarize, we can obtain a proof of global commutativity from Lemma 6.4 by showing three things:

- Assign indices to reductions in some way.
- Show that all cases other than case (2) are decreasing diagrams.
- Show that in case (2), the indices of the reductions in $c \leq^* a$ (the right completing edge) are strictly less than the indices of the spanning edges, in which event case (2) is also a decreasing diagram.

6.6.3 Almost, but not quite

Since $c \leq^* a$ is a premise of the reduction on the bottom edge, there is an obvious definition of "index" that has the property we want. The indices of $c \leq^* a$ are strictly smaller than the index of the bottom edge if we define the index of a reduction to be the depth of its derivation tree.

Unfortunately, this definition of "index" does not work, because the diagrams used in the proof of Theorem 6.1 (confluence of $\xrightarrow{\equiv}$) do not preserve depth. Since $\xrightarrow{\equiv}$ implies $\xrightarrow{\leq}$, those diagrams are cases that we must consider.

For example, consider the reduction $(\lambda x \leq a.\ b)(c) \xrightarrow{\equiv} (\lambda x \leq a.\ b)(c')$. Contracting the left redex requires $c \leq^* a$ as a premise, while contracting the right requires $c' \leq^* a$. The reduction $c \xrightarrow{\equiv} c'$ essentially becomes incorporated into the subtyping judgement used on the right, and that alters its depth.

6.7 Discussion of the proof

The proof of transitivity elimination for System $\lambda_{\triangleleft}$ has two parts: (1) confluence of $\xrightarrow{\equiv}$, and (2) commutativity of $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$. When each part is considered separately, there is a proof technique that naturally applies. Confluence of $\xrightarrow{\equiv}$ is by induction on

the number of simultaneous reductions (i.e. the diamond property). Commutativity is by induction on the index (or depth) of reductions, using the technique of decreasing diagrams.

Unfortunately, we are unable to combine these two induction principles into a single proof. The diagrams for $\xrightarrow{\equiv}$ do not preserve depth, and the diagrams for $\xrightarrow{\leq}$ do not preserve the number of simultaneous reductions. A complete proof of commutativity thus requires a stronger induction principle. Strong normalization would provide such a principle, but System $\lambda_{\lhd}$ suffers from Girard's paradox, and is not strongly normalizing.

In the absense of a suitable induction principle, it makes sense to look for counter-examples to commutativity. A counter-example would produce an infinite tiling by generating a sequence of subtype derivations that never decrease in size. Constructing such a counter-example in System $\lambda_{\lhd}$ itself would be extremely difficult, because an infinite tiling implies an infinite reduction sequence [22]. The only known infinite reduction sequences arise from Girard's paradox, and involve terms that are so large as to defy easy analysis [17].

Practical applications of pure subtype systems (e.g. first-class recursive modules) introduce fixpoints to the language, which make the construction of non-terminating expressions much easier. Nevertheless, even with fixpoints, we have been unable to construct a counter-example. The partial proof of commutativity given here is "close enough" to a full proof that all of the obvious possibilities can be eliminated. Since each half of the proof has a valid induction principle when considered on its own, any counter-example would have to exploit some subtle interaction between the two halves.

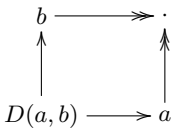
7. An open problem

One benefit of formulating the subtype relation as an abstract reduction system is that there there is an extensive literature on ARSs which could potentially be applied. Unfortunately, subtype reduction in System $\lambda_{\lhd}$ has a number of features that make it difficult to study. To simplify matters, we have devised the following rewrite system, which demonstrates confluence behavior that is very similar to subtype reduction in System $\lambda_{\lhd}$, but is formulated as a conventional ARS.

Terminals: A Non-terminals: C, D
Let $=$ be the symmetric and transitive closure of $\longrightarrow$.

A	$\longrightarrow$	$C(A)$	
$D(x, y)$	$\longrightarrow$	x	if $x = y$
$D(x, y)$	$\longrightarrow$	y	if $x = y$

The above system is a conventional, first-order conditional rewrite system, without variables, contexts, or other complications. Much like subtype reduction, it gives rise to the following diagram of local confluence:



The condition on $D(a, b) \longrightarrow a$ tells us that $a = b$. Given an appropriate induction hypothesis, we could transform $a = b$ into a transitivity-free form, giving us $a \longrightarrow\!\!\!\!\!\rightarrow \cdot \twoheadleftarrow\!\!\!\!\!\longleftarrow b$, which are the completing edges of the diagram. Notice that we complete the diagram by using the condition on one of the rewrite rules, in exactly the same way as in Section 6.6.2.

We conjecture that if a proof of confluence can be derived for the above rewrite system, then that proof can be adapted to show that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute in System $\lambda_{\lhd}$. Moreover, if a counter-example to confluence can be derived for the above rewrite system, then that counter-example can also be adapted to disprove commutativity, and consequently type safety, for System $\lambda_{\lhd}$.

8. History and Related Work

Transitivity elimination for systems with higher-order subtyping is known to be a hard problem. The first version of System $F^{\omega}_{\leq}$, developed by Steffen and Pierce [27], has an unusual restriction. In their version, polymorphic functions use bounded quantification; they are written as $\lambda X \leq T.\ u$, where T is a type, and u is an object. Type operators, however, still use kinding; they are written as $\lambda X : K.\ U$, rather than $\lambda X \leq T.\ U$. This decision breaks the symmetry of the language, and decreases its expressiveness.

The reason Steffen and Pierce made the restriction is because it simplifies the meta-theory. As they write in a footnote: "The more general form of this property... would be much more difficult to prove" [27]. Compagnoni makes the same decision in her work [10], and Chen does the same thing when adding subtyping to the calculus of constructions [8] [9].

Zwanenburg's theory of subtyping for Pure Type Systems [32] is one of the closest theories in the literature to System $\lambda_{\lhd}$. Zwanenburg's theory includes both bounded quantification and kinded quantification. Although this seems like an unnecessary duplication of syntax, there is a subtle reason for including both forms. Zwanenburg only allows higher-order subtyping (i.e. point-wise subtyping) on the kinded operators; operators with bounded quantification only have trivial subtypes.

Both Steffen and Pierce's restriction, and Zwanenburg's restriction prevent bounded quantification from being combined with higher-order subtyping. If this combination is prohibited, then confluence diagram (2) in Section 6.6.1 does not arise, because x cannot be promoted to a. As discussed previously, this particular case is what causes the diamond property of $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ to fail. If the diamond property did not fail, then we would have a proof that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute, that transitivity is admissible, and that System $\lambda_{\lhd}$ is sound.

To our knowledge, the only type theory in the literature which successfully combines higher order subtyping and bounded quantification is the version of System $F^{\omega}_{\leq}$ presented by Compagnoni and Goguen [11]. They use a proof technique called "typed operational semantics", in which every judgment comes equipped with a proof that the terms in question have normal forms. This technique can only be applied to languages that are strongly normalizing.

The proof techniques introduced in this paper demonstrate why strong normalization is useful. In our proof, we are able to show that subtype reductions commute locally, but we have been unable to show that subtype reductions commutate globally. According to Newman's lemma, however, any system which is both locally confluent (or locally commutive) and strongly normalizing is globally confluent (or commutative) [6].

By formulating the subtype relation as an abstract reduction system, we have a developed a general framework in which the results of other theories can be compared. Every other theory of subtyping in the literature has a restriction that, if applied to System $\lambda_{\lhd}$, would cause our proof technique to succeed as well.

8.1 Power types

Much of the theory of higher-order subtyping, and subtyping with dependent types, was inspired by Cardelli's early work on *power types* [7]. The approach take by Cardelli is almost completely dual to the one that we have pursued here. Rather than treating typing as a special case of subtyping, power types allow subtyping to be treated as a special case of typing. Nevertheless, the overall effect is very similar to pure subtype systems, both in terms of expressive power, and in the complexity (and intractability) of the meta-theory.

Cardelli gives the theory of power types, but not the meta-theory. Aspinall has since examined the meta-theory in more detail, but was unable to prove type safety [2]. Aspinall notes that the fundamental problem is that it is hard to prove a generation (a.k.a. inversion) lemma for power types, the same problem that we have in System $\lambda_{\triangleleft}$.

8.2 Singleton types

Aspinall has also studied the combination of subtyping with singleton types [1]. From our point of view, the most interesting thing about subtyping with singletons is that it highlights yet another symmetry between typing and subtyping. If $\{t\}$ denotes the type of all terms which are equal to t, then $\{t\} \leq T$ if and only if $t : T$. Aspinall explicitly notes that because of this symmetry, any typing judgement can be formulated as a subtype judgement, and vice versa [1].

9. Conclusion

It is tempting to conclude that the meta-theoretic difficulties which plague System $\lambda_{\triangleleft}$ stem from the fact that it unifies types and objects. However, that conclusion would be incorrect. The problems found in System $\lambda_{\triangleleft}$ can be found in any type system which has the following three elements:

1. Type operators with bounded quantification.
2. Higher order (i.e. point-wise) subtyping.
3. A language of types which is not strongly normalizing.

Moreover, there are solid practical reasons for wishing to combine these three elements together. For example, System $F^{\omega}_{\leq}$ is widely used to model inheritance in object-oriented programming languages [12], and bounded quantification is now standard in languages like Java and C#. General-purpose OO languages have fixpoints. Adding fixpoints by themselves to the level of objects in System $F^{\omega}_{\leq}$ does not create any problems, because types are completely separate from objects. However, if one were to add both fixpoints and dependent types, then the existing proof of type safety for System $F^{\omega}_{\leq}$ would break down, because dependent types are not strongly normalizing in the presence of fixpoints.

This is an important result, because a number of researchers are interested in adding dependent types and singleton types to both object-oriented and functional languages. In OO languages, dependent types are used for virtual types [20] [24], and virtual clases [15]. In functional languges, they are used for modules [13] [28].

We predict that future research will either face the same meta-theoretic difficulties that plague System $\lambda_{\triangleleft}$, or be forced to make certain compromises in the type theory, as previous work in this area has done.

References

[1] David Aspinall. Subtyping with singleton types. *Eighth International Workshop on Computer Science Logic*, 1994.

[2] David Aspinall. Subtyping with power types. *In Proceedings of Computer Science Logic*, pages 156–157, 2000.

[3] David Aspinall and Adriana Compagnoni. Subtyping dependent types. *Proceedings of 11th Annual Symposium on Logic in Computer Science*, 1996.

[4] Henk Barendregt. Lambda calculi with types. *Handbook of Logic in Computer Science, volume II*, 1992.

[5] Jan Bergstra and Jan Klop. Conditional rewrite ruels: Confluence and termination. *Journal of Computer and System Sciences*, 1986.

[6] Marc Bezem, Jan Willem Klop, and editors Roel de Vrijer. *Term Rewriting Systems*. Number 55. Cambridge Tracts in Theoretical Computer Science, 2003.

[7] Luca Cardelli. Structural subtyping and the notion of power type. *Proceedings of POPL*, 1988.

[8] Gang Chen. Subtyping calculus of constructions (extended abstract). *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, 1997.

[9] Gang Chen. Dependent type system with subtyping: Type level transitivity elimination. *Journal of Computer Science and Technology*, 14(1), 1999.

[10] Adriana Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, University of Nijmegen, 1995.

[11] Adriana Compagnoni and Healfdene Goguen. Typed operational semantics for higher order subtyping. *Information and Computation*, 184(2):242–297, 2003.

[12] Adriana Compagnoni and Benjamin Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[13] Derek Dreyer, Robert Harper, and Karl Crary. Toward a practical type theory for recursive modules. *Technical Report CMU-CS-01-112*, 2001.

[14] Derek Dreyer and Andreas Rossberg. Mixin' up the ml module system. *Proceedings of the International Conference on Functional Programming (ICFP)*, 2008.

[15] Erik Ernst, Klaus Ostermann, and William Cook. A virtual class calculus. *Proceedings of POPL*, 2006.

[16] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, 1989.

[17] Douglas Howe. The computational behavior of girard's paradox. *Proceedings of the Symposium on Logic in Computer Science*, 1987.

[18] DeLesley Hutchins. Eliminating distinctions of class: Using prototypes to model virtual classes. *Proceedings of OOPSLA*, 2006.

[19] DeLesley Hutchins. *Pure Subtype Systems: A Type Theory for Extensible Software*. PhD thesis, University of Edinburgh, 2009.

[20] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. *Proceedings of ECOOP*, 1999.

[21] Fairouz Kamareddine. Typed λ-calculi with one binder. *Journal of Functional Programming*, 15(5), 2005.

[22] Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. A geometric proof of confluence by decreasing diagrams. *Journal of Logic and Computation*, 10(3), 2000.

[23] Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994.

[24] Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A nominal theory of objects with dependent types. *Proceedings of ECOOP*, 2003.

[25] Benjamin Pierce. Bounded quantification is undecidable. *Information and Computation*, pages 131–165, 1994.

[26] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[27] Martin Steffen and Benjamin Pierce. Higher-order subtyping. *University of Edinburgh Technical Report ECS-LFCS-94-280*, 1994.

[28] Christopher Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, 2000.

[29] Masako Takahashi. Parallel reductions in λ-calculus. *Informationa and Computation*, 118(1):120–127, 1995.

[30] Vincent van Oostrom. Confluence by decreasing diagrams. *Theoretical Computer Science*, 126(1), 1994.

[31] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 2004.

[32] Jan Zwanenburg. Pure type systems with subtyping. *International Conference on Typed Lambda Calculi and Applications*, 1999.