Subtyping Dependent Types
(summary)

David Aspinall
LFCS, Department of Computer Science,
University of Edinburgh, U.K.
<David.Aspinall@dcs.ed.ac.uk>.

Adriana Compagnoni
University of Cambridge Computer Laboratory,
Cambridge, U.K.
<Adriana.Compagnoni@cl.cam.ac.uk>.

Abstract

The need for subtyping in type-systems with dependent types has been realized for some years. But it is hard to prove that systems combining the two features have fundamental properties such as subject reduction. Here we investigate a subtyping extension of the system λP*, which is an abstract version of the type system of the Edinburgh Logical Framework LF. By using an equivalent formulation, we establish some important properties of the new system* $\lambda P_{\leq}$*, including subject reduction. Our analysis culminates in a complete and terminating algorithm which establishes the decidability of type-checking.*

1. Introduction

Subtyping captures concepts from diverse areas of computer science. If A and B are sets, then $A \leq B$ ('A is a subtype of B') means that elements of A are also elements of B. If A and B are specifications, then programs satisfying specification A also satisfy B. In object-oriented programming, if A and B are object descriptions, then $A \leq B$ states that where an object with interface B is expected, it is safe to use an object with interface A. If A and B are theorems, then a proof of A is also a proof of B. Understanding the essence, subtleties, and general properties of subtyping illuminates a wide area.

Dependent types are types which depend on terms. A typical example is $\mathit{List}(n)$, the type of lists of length n. Dependent types are more expressive than simple types: the functional map can be given the type $\pi n{:}\mathit{Nat}.\mathit{List}(n) \to \mathit{List}(n)$, expressing that it is parametric in the length of lists it is applied to. More generally, type dependency can express a relationship between the input of a function and its output, which can be used to specify its behaviour. Dependent types also facilitate the encoding of logics via the *judgements-as-types* paradigm of the Edinburgh Logical Framework LF [12]. Suppose p is a term which encodes a formula of some logic. Then the dependent type $\mathit{True}(p)$ corresponds to a truth judgement and its elements encode proofs of p. The encoded proofs are constructed from constants that encode the axioms and rules of the logic.

There are several application areas where researchers have discovered a need to combine subtyping and dependent types. In the next section we give an overview of these applications; here we sketch a specific example of logic representation. (We assume some familiarity with LF; another example describing datatypes for a programming language is mentioned in Section 2.) The example is a formal system for the call-by-value λ-calculus, taken from [4].

The syntax of the call-by-value λ-calculus is the same as that of the traditional λ-calculus, but it has a restricted rule of β-equality:

$$(\lambda x.M)\ N = M[x := N] \qquad \text{provided } N \text{ is a value}$$

where a *value* is a variable or an abstraction. The restriction is achieved in LF by massaging the syntax of the encoded λ-terms. Two syntactic categories are declared:

$$\begin{array}{ll} o & : \star \\ v & : \star \end{array}$$

(these are types in LF; $\star$ is the kind of types).

The intention is that o is the type of all expressions whilst v is a subset of o corresponding to the expressions which are values. The λ-constructor, *lda*, binds terms of type v and such terms can only be variables or other terms constructed with *lda*. An extra constructor "!" is needed, which can be thought of as an injection function from values to expressions:

$$\begin{array}{ll} ! & : v \to o \\ \mathit{lda} & : (v \to o) \to v \\ \mathit{app} & : o \to o \to o \end{array}$$

For the proof system, there is an equality judgement together with constants representing axioms and rules:

$$
\begin{array}{ll}
= & : o \to o \to \star \\
E_{refl} & : \prod_{x:o} x = x \\
\vdots & \\
E_\beta & : \prod_{m:v\to o, n:v} app\ (!(lda\ m))\ (!n) = mn
\end{array}
$$

But the injection function "!" is a big nuisance. It pervades the encoding of terms yet it corresponds to nothing in the original syntax. Lambda expressions become more difficult to read and write; the example mechanisation in *LEGO* given in [4] is testimony to this. Clearly when we use the encoding we would rather not mention the injection at all.

With subtyping, we simply declare v as a subtype of o:

$$
\begin{array}{ll}
o & : \star \\
v & \leq o : \star
\end{array}
$$

and then the injection function is not needed. In effect, it becomes *implicit*: we may imagine that it is inserted automatically wherever necessary. The β-rule now reads:

$$
E_\beta \quad : \prod_{m:v\to o, n:v} app\ (lda\ m)\ n = mn
$$

and we use the same constructors as in the original syntax.

1.1 Summary of application areas

The need for subtyping in a dependently typed lambda calculus was noticed during the Edinburgh LF project, around 1987. Mason pointed out that subtypes would be useful when representing Hoare's logic: one would like to treat the type of quantifier-free boolean expressions (used in programs) as a subtype of the type of first-order formulae (used in assertions), because formulae contain quantifiers that cannot appear in programs [15]. Without subtypes extra machinery is necessary, either encoding explicit coercion functions or additional judgements to express syntactic properties. Either device complicates the encoding. (And as we have demonstrated above, other examples of encodings in LF from [4] also benefit from subtyping.)

Later, Pfenning gave more cases of cumbersome encodings of syntax, and proposed a solution by extending LF with *refinement types*, a restricted form of subtypes [16]. Moreover, he demonstrated that refinement types (or subtyping) can allow a limited form of proof reuse, so that one proof term proves several judgements. (This is connected with the interpretation of subtyping as intuitionistic implication explained by Longo *et al.* [13].) Pfenning proved that his system is decidable and is a conservative extension of LF; see Section 5 for comparison with our work.

Pfenning's application was the proof assistant *Elf* which implements LF. A richer type theory is implemented by the *LEGO* system, in which researchers at Edinburgh and Erlangen recently tested Pierce and Turner's subtyping model of object-orientation [19]. They extended the model to include proofs about objects and thus type-dependency. Because *LEGO* lacks subtyping, coercion functions are used, but it was found that inserting coercions quickly becomes tedious in practice. Other applications in *LEGO* are easy to find. So subtyping is urgently needed for proof assistants such as *Elf*, *LEGO*, and their relatives *NuPrl*, *Coq* and *Alf*. None yet have subtyping in the form we propose.

During the 1980's, Cardelli proposed several rich type systems for programming languages combining subtyping and type-dependency. The system in [8] is illustrated with examples of dependent datatypes and subtypings between them. At a workshop in 1986, Cardelli described ideas about type-checking techniques for these systems but at the outset accepted that the techniques would only lead to a semi-decision procedure, because of (for example) the combination of recursive types and type dependency [7]. We believe that our system is the first fragment of Cardelli's language, retaining subtyping and dependent types, to be shown to have a decidable type inference problem.

In algebraic specification, a language with subtyping and dependent types was proposed by Sannella *et al.* [20] to model formal program development in-the-large. A dependent type $\pi x{:}SP.SP'$ is a specification of a parameterised program that should map a program P satisfying SP to a program satisfying $SP'[x := P]$; subtyping models specification refinement. The investigations of Sannella *et al.* into this language were preliminary and the progress reported here has helped the continuation of their work in [3].

1.2 Combining subtyping and dependent types

In separation, subtyping and type-dependency have been well-studied. Yet their combination leads to systems that are difficult to study. For one thing, we tread close to the line of undecidability, as in Cardelli's system or the second-order system $F_{\leq}$ [18]. Of course, we would rather stay on the side of decidability — and this becomes essential for type theories where type-checking serves as proof-checking. For another thing, the typing and subtyping relations become intimately tangled, which means that tested techniques of examining subtyping in isolation no longer apply.

Let us quickly show how typing and subtyping become tangled. The archetypal rule of subtyping is *subsumption*, which allows a term of a type A to be used where one of a supertype B is expected:

$$
\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash M : B}
$$

(as usual, Γ denotes a context of assumptions about the types of variables — see Section 2 below). So the typing judgement depends on the subtyping judgement. When a system has dependent types like $List$ it must have a *kinding* rule to check that an application of a type-function to a term is

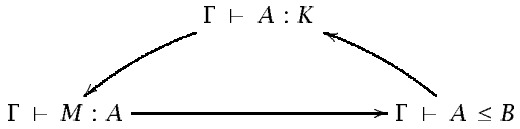
well-formed:

$$\frac{\Gamma \vdash \mathit{List} : \mathit{Nat} \to \star \qquad \Gamma \vdash n : \mathit{Nat}}{\Gamma \vdash \mathit{List}(n) : \star}$$

So the kinding judgement depends on the typing judgement. We expect the subtyping relationship to hold *a priori* only between well-formed types; for example, inferring reflexivity of subtyping between types:

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

So subtyping depends on typing, via kinding. As a picture:



This dependency turns out to significantly complicate the meta-theoretic study, compared with other well-understood subtyping systems (e.g., [11, 17, 21, 9]), which all lack this circularity.

In this paper we add subtyping to the system λP, an abstract version of the type-system (sometimes called λΠ) which underlies LF [5, 12]. This is a pure system with type-valued functions dependent on terms. In Section 2 we define $\lambda P_{\leq}$, showing examples of using the rules, and we state some basic meta-theoretic properties.

At a certain point in the development of the meta-theory, things become difficult to analyse directly. So we design an algorithmic version of the subtyping relation which breaks the cycle of dependencies. The new relation does not depend on kinding, and only relates normal forms. But still there is a circularity, since we want to know that normalization steps used by the subtyping algorithm preserve kinding. To solve this we make another separation: β-reduction is split into two levels, β_1-reduction on terms and β_2-reduction on types. Type normalization only depends on β_2-reduction; at the outset we can prove more about this than about β_1-reduction. This untangles things enough to prove equivalence of the two subtyping relations, and then properties about the original relation. This analysis of subtyping is described in Section 3.

In Section 4 we describe the type-checking algorithm. We break more dependencies between the judgements and then we prove our main result: the algorithm is correct and terminates on all inputs, so $\lambda P_{\leq}$ is decidable. A corollary is the minimal type property: every typable term possesses a minimal type in the subtype relation.

We believe that this work reports the first proof of decidability for subtyping dependent types, in a system uniformly extended with a subsumption rule and a subtyping relation. In Section 5 we summarise the achievement and compare with the related work. More details, examples and proofs can be found in a longer version of this paper [1].

2. The system $\lambda P_{\leq}$

The system $\lambda P_{\leq}$ (pronounced "lambda-pee-sub") is formally defined by the rules which follow below (also summarized at the end of the paper). The rules define four judgement forms:

$\Gamma \vdash K$	'K is a kind in context Γ'
$\Gamma \vdash A : K$	'type A has kind K in context Γ'
$\Gamma \vdash M : A$	'term M has type A in context Γ'
$\Gamma \vdash A \leq B$	'A is a subtype of B in context Γ'

For those familiar with the description of λP in [5], we differ by using a stratified presentation separating the syntactic categories of kinds, types, and terms, and replacing the start and weakening rules by the kind formation judgement. This is close to the presentation of λΠ in the appendix of [12].

The underlying grammar of *pre-terms* and *pre-contexts* is:

$$\begin{array}{rcl} M & ::= & x \mid \lambda x{:}A.M \mid MM \\ A & ::= & \alpha \mid \pi x{:}A.A \mid \Lambda x{:}A.A \mid AM \\ K & ::= & \star \mid \Pi x{:}A.K \\ \Gamma & ::= & \langle\rangle \mid \Gamma, x : A \mid \Gamma, \alpha : K \mid \\ & & \Gamma, \alpha \leq A : K \end{array}$$

We assume throughout that pre-contexts never contain repeated declarations of the same variable.

Sometimes the letters $U, V, \ldots$ will be used to range over pre-terms which may be terms, types or kinds. Substitution is defined in the usual way for term variables $U[x := M]$ and type variables $U[\alpha := A]$. As mentioned, we distinguish two kinds of β-reduction:

$$\begin{array}{rcl} C[(\lambda x{:}A.M)N] & \to_{\beta_1} & C[M[x := N]] \\ C[(\Lambda x{:}A.B)M] & \to_{\beta_2} & C[B[x := M]] \end{array}$$

($C[-]$ indicates a pre-term with a hole in it). The union of the two reductions is written $\to_\beta$. Generally, $\twoheadrightarrow_R$ is the reflexive and transitive closure of the reduction $\to_R$, and $=_R$ is the symmetric closure of $\twoheadrightarrow_R$. U^R denotes the R-normal form of U.

Formation, kinding and typing are as in λP (or λΠ), except that the type conversion rule is replaced by subsumption, and we allow bounded type-variables in the context.

Here are the rules for kind and context **formation**:

$$\frac{}{\langle\rangle \vdash \star} \qquad \text{(F-EMPTY)}$$

$$\frac{\Gamma \vdash A : \star}{\Gamma, x : A \vdash \star} \qquad \text{(F-TERM)}$$

$$\frac{\Gamma \vdash K}{\Gamma, \alpha : K \vdash \star} \qquad \text{(F-TYPE)}$$

$$\frac{\Gamma \vdash A : K}{\Gamma, \alpha \leq A : K \vdash \star} \qquad \text{(F-SUBTYPE)}$$

$$\frac{\Gamma, x : A \vdash K}{\Gamma \vdash \Pi x{:}A.K} \quad \text{(F-}\Pi\text{)}$$

The statement $\Gamma \vdash \star$ says that Γ is a well-formed context, avoiding the need for another judgement. We have two ways of adding type-variables α to a context: in (F-SUBTYPE) the declaration $\alpha \leq A : K$ declares α to have the kind K and to be *bounded* by the type A. In (F-TYPE) α is *unbounded* and only has a kind. This contrasts with other systems which have a "top" type $\top^K$ for each kind K, and recover unbounded type variables by assuming $\alpha \leq \top^K : K$. Since we have no direct application for top types, we steer clear of their bad behaviour: it is the top types that render the subtyping relation undecidable in $F_{\leq}$ when combined with a contravariant rule for bounded quantifiers [18].

As a brief example of using type-variable assumptions, here is a context Γ_{Bag} that expresses basic relationships about datatypes for bags and lists:

$$\begin{array}{rcl}\Gamma_{Bag} & \equiv & \mathsf{Nat} : \star,\ \mathsf{AllBags} : \star, \\ & & \mathsf{Bag} \leq \Lambda n{:}\mathsf{Nat}.\mathsf{AllBags} : \Pi n{:}\mathsf{Nat}.\star, \\ & & \mathsf{List} \leq \mathsf{Bag} : \Pi n{:}\mathsf{Nat}.\star\end{array}$$

(examples using Γ_{Bag} follow below). The idea is that $\mathsf{AllBags}$ is the type of all bags, and the dependent types $\mathsf{Bag}(n)$ and $\mathsf{List}(n)$ represent bags and lists of size n. A list of length n is also a bag of size n.

Here are the rules for **kinding**:

$$\frac{\Gamma \vdash \star \qquad \alpha \in \mathit{Dom}(\Gamma)}{\Gamma \vdash \alpha : \mathit{Kind}_\Gamma(\alpha)} \quad \text{(K-VAR)}$$

$$\frac{\Gamma, x : A \vdash B : \star}{\Gamma \vdash \pi x{:}A.B : \star} \quad \text{(K-}\pi\text{)}$$

$$\frac{\Gamma, x : A \vdash B : K}{\Gamma \vdash \Lambda x{:}A.B : \Pi x{:}A.K} \quad \text{(K-}\Lambda\text{)}$$

$$\frac{\Gamma \vdash A : \Pi x{:}B.K \qquad \Gamma \vdash M : B}{\Gamma \vdash AM : K[x := M]} \quad \text{(K-APP)}$$

$$\frac{\Gamma \vdash A : K \quad \Gamma \vdash K' \quad K =_\beta K'}{\Gamma \vdash A : K'} \quad \text{(K-CONV)}$$

$\mathit{Dom}(\Gamma)$ denotes the set of variables declared in Γ; in (K-VAR), $\mathit{Kind}_\Gamma(\alpha)$ refers to the kind in the declaration of α inside Γ. The kind $\star$ classifies ordinary types; the kind $\Pi x{:}A.K$ classifies dependent types.

Here are the rules for **typing**:

$$\frac{\Gamma \vdash \star \qquad x \in \mathit{Dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : \pi x{:}A.B} \quad \text{(T-}\lambda\text{)}$$

$$\frac{\Gamma \vdash M : \pi x{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash M : B} \quad \text{(T-SUB)}$$

Finally, here are the rules for **subtyping**:

$$\frac{\Gamma \vdash A : K \quad \Gamma \vdash B : K \quad A =_\beta B}{\Gamma \vdash A \leq B} \quad \text{(S-CONV)}$$

$$\frac{\Gamma \vdash A \leq B \qquad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C} \quad \text{(S-TRANS)}$$

$$\frac{\Gamma \vdash \star \qquad \alpha \text{ bounded in } \Gamma}{\Gamma \vdash \alpha \leq \Gamma(\alpha)} \quad \text{(S-VAR)}$$

$$\frac{\Gamma \vdash A' \leq A \quad \Gamma, x : A' \vdash B \leq B' \quad \Gamma \vdash \pi x{:}A.B : \star}{\Gamma \vdash \pi x{:}A.B \leq \pi x{:}A'.B'} \quad \text{(S-}\pi\text{)}$$

$$\frac{\Gamma, x : A \vdash B \leq B'}{\Gamma \vdash \Lambda x{:}A.B \leq \Lambda x{:}A.B'} \quad \text{(S-}\Lambda\text{)}$$

$$\frac{\Gamma \vdash A \leq B \qquad \Gamma \vdash BM : K}{\Gamma \vdash AM \leq BM} \quad \text{(S-APP)}$$

Conversion is included in the subtyping relation by (S-CONV), which also ensures reflexivity on types of the same kind. In (S-VAR), $\Gamma(\alpha)$ stands for the bound of α.

The rule (S-π) lets us infer subtypings such as $\pi n{:}\mathsf{Nat}.\mathsf{List}(n) \leq \pi n{:}\mathsf{Even}.\mathsf{Bag}(n)$ (when $\mathsf{Even} \leq \mathsf{Nat}$), so if we expect a function from an even number n to a bag of size n, we can use a function that maps any natural n to a list of length n.

Rules (S-Λ) and (S-APP) extend the subtyping relation to type-functions in a pointwise way. If $n : \mathsf{Nat}$, using (S-APP), (S-CONV) and (S-TRANS) we can show that $\mathsf{List}(n) \leq \mathsf{AllBags}$. Using (S-$\Lambda$) we can show that $\Lambda n{:}\mathsf{Nat}.\mathsf{List}(n) \leq \Lambda n{:}\mathsf{Nat}.\mathsf{Bag}(n)$, for example.

2.1. Basic properties of $\lambda P_{\leq}$

Many properties can be established routinely, although the order of proofs is more critical than in systems without subtyping. We can routinely prove the Church-Rosser property for β_1, β_2 or β reduction on pre-terms; strong normalization for well-formed kinds, kindable types and typable terms, and various substitution and structural properties of the judgements (see [5] or [12] for explanation and similar proofs of these properties and [1] for full details).

One desirable property of a type system is *type unicity*: the type of a term is unique up to conversion. With subtyping this cannot hold, although we can hope for the existence

of minimal types. This property is useful because it allows us to factor the problem of type-checking into two parts: the inference of a minimal type for a term and deciding the subtyping relation. We will prove that $\lambda P_{\leq}$ has minimal types in Section 4.

For the kinding fragment of our system, however, unicity does hold. Our first proposition is that the kind of a type is unique up to conversion. We use the observation that conversion at the kind level is particularly simple since there is no application. If $K =_R K'$ (where R is one of β, β_1, β_2) then for some $n \geq 0$, $K \equiv \Pi x{:}A_1 . \ldots \Pi x{:}A_n . \star$ and $K' \equiv \Pi x{:}A'_1 . \ldots \Pi x{:}A'_n . \star$ with $A_i =_R A'_i$ for each i.

Proposition 2.1 (Unicity of Kinds).
If $\Gamma \vdash A : K_1$ *and* $\Gamma \vdash A : K_2$, *then* $K_1 =_\beta K_2$.

The next property shows some anticipated agreements between the judgements, for example, that every type inhabited by a term indeed has kind $\star$.

Proposition 2.2 (Agreement of judgements).

1. *If* $\Gamma \vdash A : K$ *then* $\Gamma \vdash K$.
2. *If* $\Gamma \vdash M : A$ *then* $\Gamma \vdash A : \star$.
3. *If* $\Gamma \vdash A \leq B$ *then for some* K, $\Gamma \vdash A, B : K$.

To prove agreement, we make use of a restricted form of the following *bound narrowing* property. Once agreement is proved, we can prove the full version of narrowing. Narrowing says that reducing types in assumptions preserves the derivability of any judgement. Informally, one can see this is true by adding an instance of subsumption or transitivity to each use of a variable rule. We write $\Gamma \vdash J$ denote an arbitrary judgement.

Proposition 2.3 (Bound narrowing).
Suppose $\Gamma \vdash A' \leq A$. *Then*

1. $\Gamma, x : A, \Gamma' \vdash J$ *implies* $\Gamma, x : A', \Gamma' \vdash J$
2. $\Gamma, \alpha \leq A : K, \Gamma' \vdash J$ *implies* $\Gamma, \alpha \leq A' : K, \Gamma' \vdash J$

Another desirable property for type systems is *subject reduction*. This is the property that β-reduction preserves the type of a term. (Since a term may have several types in a subtyping system, and since an abstraction term $\lambda x{:}A.M$ may be applied to a term whose minimal type is smaller than A, in general we may have that reduction adds types.)

To prove subject reduction we need to reason about the way judgements are derived. This is the point where we hit a snag. In particular, to show that $(\lambda x{:}A.M)\,N$ and its reduct $M[x := N]$ have the same type, we would like to assume that the application was typed using (T-λ) followed by (T-APP). For this we need a *generation principle*.

Proposition 2.4 (Generation for typing).

1. *If* $\Gamma \vdash x : C$ *then* $\Gamma \vdash \Gamma(x) \leq C$.
2. *If* $\Gamma \vdash \lambda x{:}A.M : C$ *then for some* B, *(a)* $\Gamma, x : A \vdash M : B$ *and (b)* $\Gamma \vdash \pi x{:}A.B \leq C$.
3. *If* $\Gamma \vdash MN : C$ *then for some* A, B, *(a)* $\Gamma \vdash M : \pi x{:}A.B$, *(b)* $\Gamma \vdash N : A$, *and (c)* $\Gamma \vdash B[x := N] \leq C$.

Proof By induction on typing derivations, using transitivity of subtyping. □

However, this is too weak to show type preservation; the possibility that subtyping was used in (T-SUB) gets in the way. We also need a generation principle for the subtyping judgement to break down a statement of the form $\Gamma \vdash \pi x{:}A'.B' \leq \pi x{:}A.B$. Unfortunately, we cannot prove a generation principle for subtyping by induction on subtyping derivations because of the rules (S-CONV) and (S-TRANS). The next section is a quest towards generation for subtyping using a formulation without these troublesome rules.

Things are better for kinding. Generation for kinding is strong enough to prove subject reduction for β_2-reduction, which will be vital later (in Lemma 3.2).

Proposition 2.5 (Generation for kinding).

1. *If* $\Gamma \vdash \alpha : K$ *then* $K =_\beta \mathit{Kind}_\Gamma(\alpha)$.
2. *If* $\Gamma \vdash \pi x{:}A.B : K$ *then* $K \equiv \star$, *and* $\Gamma, x : A \vdash B : \star$.
3. *If* $\Gamma \vdash \Lambda x{:}A.B : K$ *then for some* K', $K =_\beta \Pi x{:}A.K'$, *and* $\Gamma, x : A \vdash B : K'$.
4. *If* $\Gamma \vdash AM : K$ *then for some* B, K', $\Gamma \vdash A : \Pi x{:}B.K'$, $\Gamma \vdash M : B$ *and* $K'[x := M] =_\beta K$.

As well as subject β_2-reduction for kinding, we also need closure of the subtyping relation under β_2-reduction. So we state a generalized form of the property, writing $J \twoheadrightarrow_{\beta_2} J'$ to indicate a β_2-reduction inside J.

Proposition 2.6 (Closure under β_2-reduction).
If $\Gamma \vdash J$ *and* $J \twoheadrightarrow_{\beta_2} J'$ *then* $\Gamma \vdash J'$.

Proof The one step case follows by induction on the derivation of $\Gamma \vdash J$ also proving the statement for a reduction inside the context Γ. The main case is an outermost reduction in (K-APP), when we need Proposition 2.5. □

3. A subtyping algorithm

To delve further into the meta-theory of $\lambda P_{\leq}$ we must confront the subtyping system. We do this by analysing an equivalent system which is *syntax directed* (to derive any given statement, at most one rule applies), and so forms an

algorithm when viewed in reverse. A generation principle for a syntax-directed system is immediate; the hard part is proving its equivalence with the original presentation.

The new rules derive statements $\Gamma \vdash_{\mathcal{A}} A \leq B$, with A and B in β_2-normal form. Normal forms allow us to grasp the fine structure of the subtyping relation, since occurrences of applications are restricted. Otherwise it is hard to tell whether an occurrence of AM was introduced by (S-APP) or (S-CONV), for example.

Our algorithm is akin to that for $F_{\wedge}^{\omega}$ in [9], with two important differences. First, the rules here have no kinding premises, so the cycle of dependencies between subtyping and typing is destroyed. Second, we make a novel adjustment for dependent types: splitting β-reduction.

We shall explain the reason for splitting β-reduction shortly. Why remove kinding premises from the subtyping rules? This was a technique used in the study of $F_{\leq}^{\omega}$ in [21], but we know from the $F_{\wedge}^{\omega}$ algorithm in [9] that removing kinding is not crucial to the study of that system. Things are more complex with $\lambda P_{\leq}$ because of the circularity between typing and subtyping: keeping kinding premises, we could reduce deciding $\Gamma \vdash_{\mathcal{A}} A \leq B$ to a finite number of typing constraints, but such constraints are in no obvious way "smaller" than the subtyping statement we began with. So it is hard to argue that an algorithm cannot loop by an infinite alternation of calls from one judgement to the other. Our first plan was to seek a cunning induction measure, but removing the circularity seems conceptually simpler and closer to a practical subtyping algorithm.

The algorithmic subtyping rules are summarized in Figure 8 at the end of the paper. The rules (AS-Π) and (AS-Λ) are the same as (S-Π) and (S-Λ) except that the kinding premises are removed and β_1-conversion of the type-label of Λ is allowed.

The other use of β_1-conversion in the algorithm is in a rule scheme corresponding to the restriction of (S-CONV) to β_2-normal forms with the shape $\alpha\, M_1 \cdots M_n$ for $n \geq 0$.

$$\frac{M_1 =_{\beta_1} M_1' \quad \cdots \quad M_n =_{\beta_1} M_n'}{\Gamma \vdash_{\mathcal{A}} \alpha\, M_1 \cdots M_n \leq \alpha\, M_1' \cdots M_n'} \qquad \text{(AS-APP-R)}$$

Transitivity is also restricted: we only allow transitivity along the bound of a type-variable in a normal form. This uses a β_2-normalization step:

$$\frac{\Gamma \vdash_{\mathcal{A}} (\Gamma(\alpha)\, M_1 \cdots M_n)^{\beta_2} \leq A}{\Gamma \vdash_{\mathcal{A}} \alpha\, M_1 \cdots M_n \leq A} \qquad \text{(AS-APP-T)}$$

It is the only place where β_2-conversion is needed. To make the rules syntax-directed, we need a side condition here that $A \neq_{\beta_1} \alpha\, M_1 \cdots M_n$, otherwise (AS-APP-R) applies.

The proof of equivalence of the two subtyping systems is split into soundness and completeness. Soundness only holds for well-kinded types, completeness only for β_2-normal forms.

In the soundness proof, subject β_2-reduction for kinding is crucial for the case of (AS-APP-T) to show that kindability is preserved from the conclusion to the premise. This is why we split up β-reduction. If the algorithmic system used β-normal forms, we would have to prove that *full* β-reduction preserved kinds — this requires subject-reduction for typing, the very thing we faltered at to begin with!

The lemma requires an auxiliary proposition.

Proposition 3.1 (Bounded Type Variables).

1. *If* $\Gamma \vdash \alpha\, M_1 \cdots M_n : K$, *then* $\Gamma \vdash \Gamma(\alpha)\, M_1 \cdots M_n : K$.
2. *If* $\Gamma \vdash \alpha\, M_1 \cdots M_n : K$, *then* $\Gamma \vdash \alpha\, M_1 \cdots M_n \leq \Gamma(\alpha)\, M_1 \cdots M_n$.

Lemma 3.2 (Soundness of algorithmic subtyping).
Suppose we have two types of the same kind, $\Gamma \vdash A, B : K$. *Then* $\Gamma \vdash_{\mathcal{A}} A \leq B$ *implies* $\Gamma \vdash A \leq B$.

Proof By induction on $\Gamma \vdash_{\mathcal{A}} A \leq B$, using structural properties and Propositions 2.3, 2.2, 2.6 and 3.1. □

For completeness we first show that reflexivity and transitivity are admissible in the new system. This is like the cut-elimination argument first used in a subtyping setting by Curien and Ghelli [11] for their study of $F_{\leq}$. But instead of showing that reflexivity and transitivity can be removed, we show that they can be added without changing the derivable statements. This avoids consideration of special "cut-free" derivations.

Proposition 3.3 (Reflexivity of algorithmic subtyping).
Let A *and* A' *be two types in* β_2 *normal-form, with* $A =_{\beta_1} A'$. *Then* $\Gamma \vdash_{\mathcal{A}} A \leq A'$.

Proof By induction on $\mathsf{size}(A) + \mathsf{size}(A')$, where $\mathsf{size}(U)$ is the number of symbols in U. Since A and A' are in β_2-normal form, their structure is identical up to term components. When A and A' are variables or applications, the result is by (AS-APP-R); when they are Π or Λ types, we use the induction hypothesis and corresponding rule. □

3.1. Admissibility of transitivity

Showing admissibility of transitivity uses extra machinery. To define a measure for the main induction, we extend the language with a new type constructor and a new reduction. The crucial property of the measure is that it reduces from the conclusion to the premises of the algorithmic subtyping rules, notably (AS-APP-T). The same measure will be used to show termination of the subtyping algorithm.

The new type constructor is a binary "plus" operator, which has the kinding rule:

$$\frac{\Gamma \vdash A : K \qquad \Gamma \vdash B : K}{\Gamma \vdash A + B : K} \qquad \text{(K-+)}$$

The idea is this. Subtyping bounded type variables α typically, but not necessarily, can involve using transitivity along the bound: $\alpha \le \Gamma(\alpha) \le D$. A type thus contains many "choice" points where the bound of a variable may or may not be used during subtyping. We define an operation $plus_\Gamma(C)$ which expands these points by recursively replacing bounded variables α in a type C with $\alpha + \Gamma(\alpha)$.

We can recover a plus-free type from $plus_\Gamma(C)$ by choosing either the left or right side of every plus expression. This is captured by $+$-reduction:

$$\begin{array}{lll} \mathsf{C}[A + B] & \longrightarrow_+ & \mathsf{C}[A] \\ \mathsf{C}[A + B] & \longrightarrow_+ & \mathsf{C}[B] \end{array}$$

(where $\mathsf{C}[-]$ is a type or term in the extended language with a hole in it). The number of $+$-reductions possible from $plus_\Gamma(C)$ affects the complexity of deciding a subtyping statement containing the type C.

Definition 3.4 (Plus-expansion of a type).
Let Γ be a context and declare all the type variables of a type C. Then $plus_\Gamma(C)$ is given by:

$$\begin{array}{rcl} plus_{\Gamma_1,\,\alpha \le A:K,\,\Gamma_2}(\alpha) & = & \alpha + plus_{\Gamma_1}(A) \\ plus_{\Gamma_1,\,\alpha:K,\,\Gamma_2}(\alpha) & = & \alpha \\ plus_\Gamma(\Pi x{:}A.B) & = & \Pi x{:}plus_\Gamma(A).\, plus_\Gamma(B) \\ plus_\Gamma(\Lambda x{:}A.B) & = & \Lambda x{:}plus_\Gamma(A).\, plus_\Gamma(B) \\ plus_\Gamma(A\,M) & = & plus_\Gamma(A)\,M \end{array}$$

When the variable condition on Γ is met, $plus_\Gamma(C)$ is defined uniquely — this can be shown by appealing to properties of contexts and observing that the definition is well-founded on the lexicographic ordering of pairs $\langle length(\Gamma),\, size(C)\rangle$, where $length(\Gamma)$ is the number of variables declared by Γ.

One important fact is that there is a $+$-reduction from the expansion of a type-variable to its bound in the context; this is used in the next proposition. We write $\twoheadrightarrow_R^n$ to indicate that a reduction is n steps long and $\twoheadrightarrow_R^{>n}$ for more than n steps. We extend β_2-reduction to $A + B$ in the obvious (compatible) way.

Proposition 3.5 (Plus types and reduction).
$plus_\Gamma(\alpha\,M_1 \cdots M_n)$
$\twoheadrightarrow_{\beta_2+}^{>0} plus_\Gamma((\Gamma(\alpha)\,M_1 \cdots M_n)^{\beta_2})$.

Proof We use several sub-lemmas to prove the statement:

1. For two contexts Γ and Γ', if $\Gamma \subseteq \Gamma'$ then $plus_\Gamma(A) = plus_{\Gamma'}(A)$. ($\Gamma \subseteq \Gamma'$ means that every declaration appearing in Γ is also in Γ').

2. $plus_\Gamma(\alpha\,M_1 \cdots M_n) \twoheadrightarrow_+^{>0} plus_\Gamma(\Gamma(\alpha)\,M_1 \cdots M_n)$

3. If $A \longrightarrow_{\beta_2} B$, then $plus_\Gamma(A) \twoheadrightarrow_{\beta_2} plus_\Gamma(B)$

Parts 1 and 3 follow by induction on the structure of A. Part 2 follows by induction on n: in the base case, we have $plus_\Gamma(\alpha) \twoheadrightarrow_+^{>0} plus_\Gamma(\Gamma(\alpha))$ by the definition of $plus$. The desired result then follows from 2 and 3. □

Now β_2+-reduction will help define the measure we seek. First, let $maxred_\Gamma(A)$ be the maximal number of β_2+-reductions from the plus-expansion of a type:

$$maxred_\Gamma(A) =_{\text{def}} max\left\{\, n \;\middle|\; plus_\Gamma(A) \twoheadrightarrow_{\beta_2+}^{n} A' \text{ for some } A' \,\right\}$$

(Notice that $maxred_\Gamma(A)$ only makes sense when $plus_\Gamma(A)$ is β_2+-strongly normalizing.) Then we define the *weight* of two types A, B as the pair:

$$weight_\Gamma(A,B) =_{\text{def}} \langle\, maxred_\Gamma(A) + maxred_\Gamma(B),\; size(A) + size(B) \,\rangle$$

The number of bounded variables and the size of the types both contribute. Pairs $weight_\Gamma(A,B)$ are well-ordered by the usual lexicographic ordering.

Proposition 3.6 (Transitivity of algorithmic subtyping).
Suppose $plus_\Gamma(A)$, $plus_\Gamma(B)$, $plus_\Gamma(C)$ and the plus-expansion of the bound of every type variable in Γ are all β_2+-strongly normalizing. Then $\Gamma \vdash_{\mathcal{A}} A \le B$ and $\Gamma \vdash_{\mathcal{A}} B \le C$ implies $\Gamma \vdash_{\mathcal{A}} A \le C$.

Proof For all Γ using induction on $weight_\Gamma(A,C)$. By the assumption and properties of normal forms the measure is always well-defined. Then using case analysis on the last rule used to derive $\Gamma \vdash_{\mathcal{A}} A \le B$ we can break down the transitivity into smaller instances, using Proposition 3.5. □

3.2. Completeness of algorithmic subtyping

Now we can establish completeness, using some properties of the new system. Parts 2 and 3 of the next proposition hold for all M such that the normal forms mentioned exist (a weaker condition than kindability).

Proposition 3.7 (Properties of algorithmic subtyping).

1. *If $\Gamma \vdash_{\mathcal{A}} A \le B$ and $\Gamma =_{\beta_2} \Gamma'$, then $\Gamma' \vdash_{\mathcal{A}} A \le B$.*

2. *If $\Gamma_1, x : A, \Gamma_2 \vdash_{\mathcal{A}} B \le C$, then $\Gamma_1, \Gamma_2[x := M] \vdash_{\mathcal{A}} (B[x := M])^{\beta_2} \le (C[x := M])^{\beta_2}$.*

3. *If $\Gamma \vdash_{\mathcal{A}} A \le B$ and B is not a Π-type, then $\Gamma \vdash_{\mathcal{A}} (AM)^{\beta_2} \le (BM)^{\beta_2}$.*

Part 3 is crucial in the completeness proof, where the induction hypothesis alone is too weak to show the admissibility of (S-APP).

We can prove that β_2+-reduction is strongly normalizing on well-kinded types in the language extended with $+$.

Moreover if $\Gamma \vdash A : K$ in the language without +, then $\Gamma \vdash plus_\Gamma(A) : K$ in the extended language. So whenever types A, B, and C are kindable in a context Γ, the condition of Proposition 3.6 is satisfied.

Lemma 3.8 (Completeness of algorithmic subtyping). *If* $\Gamma \vdash A \leq B$ *then* $\Gamma \vdash_{\mathcal{A}} A^{\beta_2} \leq B^{\beta_2}$.

Proof Induction on derivations, using Propositions 2.2, 3.6 and 3.7. □

Equivalence of the two systems gives a powerful tool for analysing the subtyping relation. We can prove the generation principle we wanted.

Proposition 3.9 (Generation for subtyping).

1. *If* $\Gamma \vdash \alpha \leq C$ *and* α *is bounded in* Γ, *then either* $C =_\beta \alpha$, *or* $\Gamma \vdash \Gamma(\alpha) \leq C$.
2. *If* $\Gamma \vdash \Pi x{:}A.B \leq C$ *then for some* A', B', *(a)* $C =_\beta \Pi x{:}A'.B'$, *(b)* $\Gamma \vdash A' \leq A$, *and (c)* $\Gamma, x : A' \vdash B \leq B'$.
3. *If* $\Gamma \vdash \Lambda x{:}A.B \leq C$ *then for some* B', *(a)* $C =_\beta \Lambda x{:}A.B'$, *and (b)* $\Gamma, x : A \vdash B \leq B'$.

Proof Using Lemma 3.8, by considering the last rule of a derivation in the algorithmic system and then converting back to the original system using Lemma 3.2. □

And finally, the subject reduction property for β_1-reduction.

Proposition 3.10 (Closure under β_1-reduction). *If* $\Gamma \vdash J$ *and* $J \twoheadrightarrow_{\beta_1} J'$ *then* $\Gamma \vdash J'$

Proof As for Proposition 2.6, except the case of an outermost reduction is in the rule (T-APP), where we use generation for both typing and subtyping. □

4. A type-checking algorithm

Our next step towards proving decidability is to design algorithmic versions of the remaining judgements. In the same way that we removed kinding premises from subtyping, we remove formation premises from kinding and typing. Again this gives us something nearer a feasible algorithm, and helps prove termination.

Figures 2, 4 and 6 at the end of the paper show the new rules against the old ones, below we just give highlights. With the convention that premises are evaluated in order (from left to right, 'stacked' premises from top to bottom), the rules form an algorithm.

For formation, the rule for introducing a bounded type-variable becomes:

$$\frac{\Gamma \vdash_{\mathcal{A}} K \qquad \Gamma \vdash_{\mathcal{A}} A : K' \qquad K =_\beta K'}{\Gamma, \alpha \leq A : K \vdash_{\mathcal{A}} \star} \quad \text{(AF-SUBTYPE)}$$

The first premise checks the well-formedness of the new kind K and the context. The second premise finds a kind K' for the bound A (which we expect to be a well-formed kind). Then it is safe to check that the kinds are convertible by normalizing. Conversion is needed because it has been removed from algorithmic kinding to make the rules syntax-directed.

The algorithmic rule for kinding applications is:

$$\frac{\Gamma \vdash_{\mathcal{A}} A : \Pi x{:}B.K \qquad \begin{array}{l}\Gamma \vdash_{\mathcal{A}} M : B' \\ \Gamma \vdash_{\mathcal{A}} B'^{\beta_2} \leq B^{\beta_2}\end{array}}{\Gamma \vdash_{\mathcal{A}} A\,M : K[x := M]} \quad \text{(AK-APP)}$$

The subtyping premise is necessary because subsumption is removed from the new typing relation. Similarly, we allow subtyping when typing term applications:

$$\frac{\begin{array}{ll}\Gamma \vdash_{\mathcal{A}} M : A & \Gamma \vdash_{\mathcal{A}} N : B' \\ FLUB_\Gamma(A) \equiv \Pi x{:}B.C & \Gamma \vdash_{\mathcal{A}} B'^{\beta_2} \leq B\end{array}}{\Gamma \vdash_{\mathcal{A}} MN : C[x := N]} \quad \text{(AT-APP)}$$

The function $FLUB$ ("functional least upper bound") is used to find a Π-type for the type of a term. It climbs the context, following the subtyping order, until it finds a Π-type or can go no farther. This is achieved by repeatedly β_2-normalizing and replacing head variables by their bounds.

$$FLUB_\Gamma(A) = \begin{cases} FLUB_{\Gamma|_\alpha}(\Gamma(\alpha)\,M_1 \cdots M_n) & A^{\beta_2} \equiv \alpha\,M_1 \cdots M_n \\ A^{\beta_2} & \text{otherwise} \end{cases}$$

(where the first case only applies if α is declared with a bound in Γ, and then $\Gamma|_\alpha$ is the initial prefix of Γ up to the declaration of α).

Now we must show that the new algorithmic rules are sound and complete with respect to the original ones. This is easier than it was for subtyping, making use of some properties of $FLUB$, including the fact that $FLUB_\Gamma(A)$ is an upper bound of A.

Proposition 4.1 (Properties of $FLUB$).

1. *If* $\Gamma \vdash A : K$, *then* $FLUB_\Gamma(A)$ *is well-defined.*
2. *If* $FLUB_\Gamma(A)$ *is defined and* Γ *is a prefix of the context* Γ', *then* $FLUB_{\Gamma'}(A) \equiv FLUB_\Gamma(A)$.
3. *If* $\Gamma \vdash A : K$, *then* $\Gamma \vdash A \leq FLUB_\Gamma(A)$.

Lemma 4.2 (Soundness of algorithmic system). *For all* Γ, A, K, M,

1. $\Gamma \vdash_{\mathcal{A}} K$ *implies* $\Gamma \vdash K$,
2. *If* $\Gamma \vdash \star$ *then* $\Gamma \vdash_{\mathcal{A}} A : K$ *implies* $\Gamma \vdash A : K$,
3. *If* $\Gamma \vdash \star$ *then* $\Gamma \vdash_{\mathcal{A}} M : A$ *implies* $\Gamma \vdash M : A$.

Proof Simultaneously by induction on the derivation in the algorithmic system. Proposition 4.1 is needed for (AK-APP) and Lemma 3.2 is needed for (AT-APP). □

For completeness we use the crucial characteristic of FLUB, which justifies its name: if a type A is a subtype of some Π-type, then $\mathsf{FLUB}_\Gamma(A)$ is the least Π-type greater than or equal to A in the subtype ordering. So an application typed with (AT-APP) is given a minimal type.

Proposition 4.3 (Π-types and FLUB).

1. *If* $\Gamma \vdash A \leq \Pi x{:}C.D$ *then* $\mathsf{FLUB}_\Gamma(A) \equiv \Pi x{:}C'.D'$ *for some* C', D'.
2. *If* $\Gamma \vdash A \leq \Pi x{:}C.D$ *then* $\Gamma \vdash \mathsf{FLUB}_\Gamma(A) \leq \Pi x{:}C.D$.

Proof Via corresponding results for $\vdash_{\mathcal{A}}$. □

Lemma 4.4 (Completeness of algorithmic system).

1. *If* $\Gamma \vdash K$, *then* $\Gamma \vdash_{\mathcal{A}} K$.
2. *If* $\Gamma \vdash A : K$, *then there is a* K_a *such that* $\Gamma \vdash_{\mathcal{A}} A : K_a$, $K_a =_\beta K$ *and* $\Gamma \vdash K_a$.
3. *If* $\Gamma \vdash M : A$, *then there is an* A_a *such that* $\Gamma \vdash_{\mathcal{A}} M : A_a$ *and* $\Gamma \vdash A_a \leq A$.

Proof Simultaneously by induction on derivations in the original system, using the corresponding algorithmic rules, structural properties, Propositions 2.2 and 4.3. □

This also proves that the algorithmic typing rules assign a minimal type to a typable term. The minimal typing property for the original system follows, by soundness.

Corollary 4.5 (Minimal typing property for $\lambda P_\leq$).
Whenever $\Gamma \vdash M : A$, *then there is an* A_a *such that* $\Gamma \vdash M : A_a$ *and* $\Gamma \vdash A_a \leq A$.

Our final theorem establishes the decidability of the algorithmic judgements, which guarantees the termination of subtype checking, kind inference, minimal type inference and formation checking.

Theorem 4.6 (Decidability).
For all Γ, K, M, A, *and* B, *the following problems are decidable:*

1. *Given* K *and* K' *such that* $\Gamma \vdash A : K$ *and* $\Gamma \vdash B : K'$, *whether* $\Gamma \vdash_{\mathcal{A}} A \leq B$.
2. *Provided* $\Gamma \vdash \star$, *whether there exists a* K_a *such that* $\Gamma \vdash_{\mathcal{A}} A : K_a$.
3. *Provided* $\Gamma \vdash \star$, *whether there exists an* A_a *such that* $\Gamma \vdash_{\mathcal{A}} M : A_a$.
4. *Whether* $\Gamma \vdash_{\mathcal{A}} K$.

Proof Part 1 by induction on $\mathsf{weight}_\Gamma(A, B)$. Parts 2 and 3 using part 1 by simultaneous induction on the size of term to the left of ":", and part 4 by induction on the size of the judgement, using parts 1–3. □

Lemmas 3.2, 3.8, 4.2 and 4.4 establish an equivalence between the two presentations of $\lambda P_\leq$. Put together with Theorem 4.6, this equivalence shows that we have a correct and terminating algorithm for deciding any judgement.

To see how the algorithmic rules yield an algorithm, consider the subtyping judgement. To check if $\Gamma \vdash A \leq B$, we first check that $\Gamma \vdash_{\mathcal{A}} \star$. Knowing this, we find whether A and B have kinds, say $\Gamma \vdash_{\mathcal{A}} A : K_a$ and $\Gamma \vdash_{\mathcal{A}} B : K_b$. This means that A and B are normalizing, so we can normalize them and finally check if $\Gamma \vdash_{\mathcal{A}} A^{\beta_2} \leq B^{\beta_2}$. If any step fails, then $\Gamma \vdash A \leq B$ does not hold.

The other judgements yield similar procedures. As stated, these are of theoretical interest only; we expect that practical implementations would make use of β_2 weak-head normal forms instead of full normal forms, amongst other efficiency improvements.

5. Conclusion

Our system $\lambda P_\leq$ adds subtyping to λP. The system λP is the simplest corner of Barendregt's λ-cube with type dependency, yet it is the core of applied type-theories for which subtyping is desirable. Subtyping posed a challenge for meta-theoretical study; we met the challenge by proving properties in a carefully chosen order and formulating an algorithmic version of the system. The main result is the decidability of the typing and subtyping relations, achieved using non-trivial extensions of work that dates back to Cardelli's early ideas [7], Curien and Ghelli's analysis of $F_\leq$ [11] and subsequent studies of non-dependent subtyping systems [17, 9, 21].

Of the related work, Pfenning's study of refinement types [16] is closest. There, a *sort* is declared as a *refinement* of a type, and there is a *subsorting* relation. Whilst subsorting is a richer relation than our subtyping (for example, intersections of sorts are permitted), there is a strict separation between types and sorts to ensure a straightforward proof of decidability of the system. Sorts cannot appear in labels of λ-abstractions, so it is impossible to write functions with domains limited via subsorting, a disadvantage Pfenning mentions. No such restriction applies to our calculus, where subtyping applies uniformly.

Other related work includes that of Cardelli [7, 8], who gave basic definitions and ideas about semi-decision procedures; Aspinall [2], who describes a system that has subtyping and dependent types but no type variables; Coquand [10] and Luo [14] who each consider forms of subtyping inductive data types in a dependent type-theory, and Betarte and Tasistro (Chalmers University) who recently investigated adding dependent records to Martin-Löf's type theory.

We want to continue the work begun here in several ways. The first goal is to find a semantics for $\lambda P_\leq$. The ideal would be to translate $\lambda P_\leq$ into λP by removing subtyp-

ing, along the lines of [6]. We hinted at this understanding in Section 1 when we suggested that the injection function "!" is somehow implicit in the presence of subtyping, as if inserted automatically. To generalise, we must assume families of coercions for each bounded type variable in a $\lambda P_{\leq}$ context, and show that there is a canonical way of inserting coercions to translate pre-terms at each level to λP. Then any model of λP will serve as a model of $\lambda P_{\leq}$ and the class of logics that can be encoded will be the same as for LF.

For the application of logic encoding, it is well known that including η-conversion in the framework is important. Studying examples, the need for intersection types which Pfenning recognised also seems important, allowing constants to be overloaded. If the techniques of [9] can be adapted, we could reproduce Pfenning's examples in [16].

In another direction, we need to examine richer type systems, including the polymorphism and bounded quantification of $F_{\leq}$, and approaching the type theories underlying the proof assistants mentioned in the introduction. We suspect that a careful combination of these features would also give a good type system for a programming language, although investigation of programming with type-dependency alone is in its infancy. And to integrate our work into real proof assistants, we must consider more than type-checking, since systems like *Elf* and *LEGO* do more than check proofs. Searching for a proof or applying a tactic involves unification or matching procedures which would need modification to take subtyping into account.

Finally, to curb a series of papers on new subtyping systems, it would be nice to lift our results to a more general setting, pursuing the idea (which has occurred to several researchers) of adding subtyping to Pure Type Systems [5]. It is easy to formulate such extensions, maybe using Cardelli's power types [8], but it seems much harder to prove things about them. We hope that variations of the techniques used here may help.

Acknowledgements. Our sincere thanks are due to R. Constable, M. Dezani, H. Goguen, M. Hofmann, Z. Luo, R. Pollack and D. Sannella for their comments on drafts. We gratefully acknowledge the support provided by the LFCS and a UK EPSRC studentship and grant GR/H73103 (DA) and EPSRC grants GR/K38403, GR/G55792, and CONFER and EuroFOCS (AC).

References

[1] D. Aspinall and A. Compagnoni. Subtyping dependent types. LFCS technical report, Department of Computer Science, University of Edinburgh. To appear, 1996.

[2] D. R. Aspinall. Subtyping with singleton types. In *Proc. Computer Science Logic, CSL'94, Kazimierz, Poland*, Lecture Notes in Computer Science 933. Springer-Verlag, 1995.

[3] D. R. Aspinall. *Type Systems for Parameterisation and Refinement in Algebraic Specification*. PhD thesis, Department of Computer Science, University of Edinburgh, 1996. Forthcoming.

[4] A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

[5] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[6] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[7] L. Cardelli. Typechecking dependent types and subtypes. In M. Boscarol, L. C. Aiello, and G. Levi, editors, *Proc. of the Workshop on Foundations of Logic and Functional Programming*, Lecture Notes in Computer Science 306. Springer, 1987.

[8] L. Cardelli. Structural subtyping and the notion of power type. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988.

[9] A. B. Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, Nijmegen Catholic University, 1995.

[10] T. Coquand. Pattern matching with dependent types. In *Proceedings on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.

[11] P.-L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[12] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, 1993.

[13] G. Longo, K. Milsted, and S. Soloviev. A logic of subtyping (extended abstract). In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 292–299, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[14] Z. Luo. Coercive subtyping. Draft paper, Department of Computer Science, University of Durham, 1995.

[15] I. A. Mason. Hoare's Logic in the LF. Technical Report ECS-LFCS-87-32, LFCS, Department of Computer Science, University of Edinburgh, June 1987.

[16] F. Pfenning. Refinement types for logical frameworks. In *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 315–328, May 1993.

[17] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[18] B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1), July 1994.

[19] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, Apr. 1994.

[20] D. T. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

[21] M. Steffen and B. Pierce. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, June 1994.

$$\frac{}{\langle\rangle \vdash \star} \quad \text{(F-EMPTY)}$$

$$\frac{\Gamma \vdash A : \star}{\Gamma, x : A \vdash \star} \quad \text{(F-TERM)}$$

$$\frac{\Gamma \vdash K}{\Gamma, \alpha : K \vdash \star} \quad \text{(F-TYPE)}$$

$$\frac{\Gamma \vdash A : K}{\Gamma, \alpha \leq A : K \vdash \star} \quad \text{(F-SUBTYPE)}$$

$$\frac{\Gamma, x : A \vdash K}{\Gamma \vdash \Pi x{:}A.K} \quad \text{(F-}\Pi\text{)}$$

Figure 1. Formation of contexts and kinds

$$\frac{}{\langle\rangle \vdash_{\mathcal{A}} \star} \quad \text{(AF-EMPTY)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} \star \qquad \Gamma \vdash_{\mathcal{A}} A : \star}{\Gamma, x : A \vdash_{\mathcal{A}} \star} \quad \text{(AF-TERM)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} K}{\Gamma, \alpha : K \vdash_{\mathcal{A}} \star} \quad \text{(AF-TYPE)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} K \qquad \Gamma \vdash_{\mathcal{A}} A : K' \qquad K =_\beta K'}{\Gamma, \alpha \leq A : K \vdash_{\mathcal{A}} \star} \quad \text{(AF-SUBTYPE)}$$

$$\frac{\Gamma, x : A \vdash_{\mathcal{A}} K}{\Gamma \vdash_{\mathcal{A}} \Pi x{:}A.K} \quad \text{(AF-}\Pi\text{)}$$

Figure 2. Algorithmic Formation

$$\frac{\Gamma \vdash \star \qquad \alpha \in Dom(\Gamma)}{\Gamma \vdash \alpha : Kind_\Gamma(\alpha)} \quad \text{(K-VAR)}$$

$$\frac{\Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi x{:}A.B : \star} \quad \text{(K-}\Pi\text{)}$$

$$\frac{\Gamma, x : A \vdash B : K}{\Gamma \vdash \Lambda x{:}A.B : \Pi x{:}A.K} \quad \text{(K-}\Lambda\text{)}$$

$$\frac{\Gamma \vdash A : \Pi x{:}B.K \qquad \Gamma \vdash M : B}{\Gamma \vdash AM : K[x := M]} \quad \text{(K-APP)}$$

$$\frac{\Gamma \vdash A : K \qquad \Gamma \vdash K' \qquad K =_\beta K'}{\Gamma \vdash A : K'} \quad \text{(K-CONV)}$$

Figure 3. Kinding

$$\frac{\alpha \in Dom(\Gamma)}{\Gamma \vdash_{\mathcal{A}} \alpha : Kind_\Gamma(\alpha)} \quad \text{(AK-VAR)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} A : \star \qquad \Gamma, x : A \vdash_{\mathcal{A}} B : \star}{\Gamma \vdash_{\mathcal{A}} \Pi x{:}A.B : \star} \quad \text{(AK-}\Pi\text{)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} A : \star \qquad \Gamma, x : A \vdash_{\mathcal{A}} B : K}{\Gamma \vdash_{\mathcal{A}} \Lambda x{:}A.B : \Pi x{:}A.K} \quad \text{(AK-}\Lambda\text{)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} A : \Pi x{:}B.K \qquad \begin{array}{l}\Gamma \vdash_{\mathcal{A}} M : B' \\ \Gamma \vdash_{\mathcal{A}} B'^{\beta_2} \leq B^{\beta_2}\end{array}}{\Gamma \vdash_{\mathcal{A}} AM : K[x := M]} \quad \text{(AK-APP)}$$

Figure 4. Algorithmic Kinding

$$\frac{\Gamma \vdash \star \qquad x \in Dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B} \quad (\text{T-}\lambda)$$

$$\frac{\Gamma \vdash M : \Pi x{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash M : B} \quad \text{(T-SUB)}$$

Figure 5. Typing

$$\frac{x \in Dom(\Gamma)}{\Gamma \vdash_{\mathcal{A}} x : \Gamma(x)} \quad \text{(AT-VAR)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} A : \star \qquad \Gamma, x : A \vdash_{\mathcal{A}} M : B}{\Gamma \vdash_{\mathcal{A}} \lambda x{:}A.M : \Pi x{:}A.B} \quad (\text{AT-}\lambda)$$

$$\frac{\begin{array}{ll}\Gamma \vdash_{\mathcal{A}} M : A & \Gamma \vdash_{\mathcal{A}} N : B' \\ FLUB_\Gamma(A) \equiv \Pi x{:}B.C & \Gamma \vdash_{\mathcal{A}} B'^{\beta_2} \leq B\end{array}}{\Gamma \vdash_{\mathcal{A}} MN : C[x := N]} \quad \text{(AT-APP)}$$

Figure 6. Algorithmic Typing

$$\frac{\Gamma \vdash A : K \qquad \Gamma \vdash B : K \qquad A =_\beta B}{\Gamma \vdash A \leq B} \quad \text{(S-CONV)}$$

$$\frac{\Gamma \vdash A \leq B \qquad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C} \quad \text{(S-TRANS)}$$

$$\frac{\Gamma \vdash \star \qquad \alpha \text{ bounded in } \Gamma}{\Gamma \vdash \alpha \leq \Gamma(\alpha)} \quad \text{(S-VAR)}$$

$$\frac{\begin{array}{l}\Gamma \vdash A' \leq A \\ \Gamma, x : A' \vdash B \leq B' \qquad \Gamma \vdash \Pi x{:}A.B : \star\end{array}}{\Gamma \vdash \Pi x{:}A.B \leq \Pi x{:}A'.B'} \quad (\text{S-}\Pi)$$

$$\frac{\Gamma, x : A \vdash B \leq B'}{\Gamma \vdash \Lambda x{:}A.B \leq \Lambda x{:}A.B'} \quad (\text{S-}\Lambda)$$

$$\frac{\Gamma \vdash A \leq B \qquad \Gamma \vdash BM : K}{\Gamma \vdash AM \leq BM} \quad \text{(S-APP)}$$

Figure 7. Subtyping

$$\frac{\Gamma \vdash_{\mathcal{A}} A' \leq A \qquad \Gamma, x : A' \vdash_{\mathcal{A}} B \leq B'}{\Gamma \vdash_{\mathcal{A}} \Pi x{:}A.B \leq \Pi x{:}A'.B'} \quad (\text{AS-}\Pi)$$

$$\frac{A =_{\beta_1} A' \qquad \Gamma, x : A' \vdash_{\mathcal{A}} B \leq B'}{\Gamma \vdash_{\mathcal{A}} \Lambda x{:}A.B \leq \Lambda x{:}A'.B'} \quad (\text{AS-}\Lambda)$$

$$\frac{M_1 =_{\beta_1} M_1' \quad \cdots \quad M_n =_{\beta_1} M_n'}{\Gamma \vdash_{\mathcal{A}} \alpha M_1 \cdots M_n \leq \alpha M_1' \cdots M_n'} \quad \text{(AS-APP-R)}$$

$$\frac{\begin{array}{l}\alpha \text{ bounded in } \Gamma, \ A \notin \alpha M_1 \cdots M_n \\ \Gamma \vdash_{\mathcal{A}} (\Gamma(\alpha) M_1 \cdots M_n)^{\beta_2} \leq A\end{array}}{\Gamma \vdash_{\mathcal{A}} \alpha M_1 \cdots M_n \leq A} \quad \text{(AS-APP-T)}$$

Figure 8. Algorithmic Subtyping