FI MU

Faculty of Informatics
Masaryk University Brno

A Calculus of Coercive Subtyping

by

Matej Kollár
Ondřej Peterka
Ondřej Ryšavý
Libor Škarvada

FI MU Report Series **FIMU-RS-2009-11**

 November 2009

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can be obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**

A Calculus of Coercive Subtyping

Matej Kollár Ondřej Peterka Ondřej Ryšavý Libor Škarvada

December 3, 2009

Abstract

This report deals with a type system that merges subtyping and dependent types. We define a calculus, denoted as $\lambda P_{\hookrightarrow}$, that instead of term overloading employs coercion mappings. This enables to detach the subtyping from other parts of the calculus, so that the mutual dependence between subtyping, typing and kinding can be diminished. We analyze basic properties of the calculus and show several examples that demonstrate the mechanism of coercive subtyping.

1 Introduction

Subtyping appears to be a convenient tool in programming languages that allows for implementation of reuse techniques. The idea to include subtyping in frameworks of type theory emerged when the need for proof development in large was recognized as an issue, whose overcoming would help to spread tools based on type theory to practice. Subtyping dependent types was initially studied for the purpose of extending programming languages [1, 7]. Later the importance of this concept led to an intensive research on extending various type theoretical systems with subtyping, having form, for instance, of so-called refinement types [19], dependent record types [5, 6], and coercions [4, 17]. The extension of various type theory tools with subtyping has been proposed [8, 10]. Several works attempted to clarify the issue of combining subtyping and dependent types. The system λP, which is the foundation of type theoretical frameworks and a vertex of Barendregt's λ-cube, was extended with subtyping in [2], and the examination of meta-theoretical properties gave a proof of decidability of typechecking. Along this line, the idea was later extended to the calculi of the whole λ-cube in [20].

The meaning of subtyping, $A \leq B$, can be viewed from different aspects. The intuitive one, based on the set theory, says that A is a subset of B, when A and B are

considered as sets. In programming, the usual interpretation is, that it is legal to use a value of type A in a context where value of type B is expected. This is formalized by so-called subsumption rule:

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

In the context of type theory, this can be seen as a generalization of *conversion rule*, in which $\Gamma \vdash A = B$ is assumed in the premise instead of the given subtype relation. Traditionally, the subtyping relation over types is defined such that it overloads terms. Then certain properties like existence of principal or minimal typing are investigated. The more flexible approach based on coercion was later introduced to explicitly describe the transformation of terms required to relate types in subtype relation. The coercion represents a valid term in the environment that gives one a method of providing the term of a correct type as expected by the given context.

The main issue in systems combining dependent types and subtyping is to show that they have required meta-theoretical properties, mainly subject reduction and decidability of typechecking. Subtyping is tightly related to typing and introduces extra dependence among rules of the system. To break this dependence and show supporting properties, such as transitivity elimination and subtyping decidability, one needs to modify the original system to get, for instance, equivalent algorithmical system with required properties. In the present paper, we deal with typing and subtyping uniformly. We introduce a coercion also for computationally equal types with the hope of simplification of the type system and steps required to prove its fundamental properties.

Our work stems, in particular, from the research done by Aspinall and Compagnoni [2], and Luo [16]. The calculus defined in this paper is based on λP system and the shape of many type rules and the idea of splitting reduction on term and type level corresponds to $\lambda P_{\leq}$ system of Aspinall and Compagnoni. The introduced notion of coercion functions is motivated by definitions presented in [16]. Therein, a coercion κ is a definable term in the system, and is inserted if necessary to harmonize the type of expressions. It is done via introduction of abbreviations, $f(x) = f(\kappa\ a)$, for f with domain of type A' and coercion $\kappa : A \to A'$ that allows to apply the function to term $a : A$. In our work, we take coercion as definitional term that stands for a function usable for transforming terms between subtypes or computationally equivalent types. In the case of coercion for computational type equality such mapping is simply the identity function that may be removed from the term.

The contribution of the present work is in formalization of an extension of λP calculus with coercive subtyping, in which the subtyping fragment itself is more independent compared to related calculi, such as $\lambda P_{\leq}$ or λΠ [9]. In fact, the only dependence comes with the need to deal with type equality, but this is restricted in subtyping fragment. Because of this restriction, by allowing to define subtyping only on types in canonical form it is possible to break the dependence as a whole and show decidability of subtyping fragment independently. Moreover, it can be recognized that the subtyping fragment corresponds to simply typed calculus (considering $\hookrightarrow$ as a function type constructor). Contrary to related work mentioned above, we deal only with the simple form of coercive subtyping, which imposes several limitations. At the end of the paper we suggest possible extensions of the system that overcome these restrictions and provide more flexible subtyping comparable to $\lambda P_{\leq}$.

The present paper is organized as follows: In Section 2, we give a formal presentation of $\lambda P_{\hookrightarrow}$ calculus, which is supplied with examples to help grasp the idea of the system. In Section 3, we state basic properties of the calculus. In particular, we sketch the proof of decidability of typechecking. In Section 4, we conclude with discussion and remarks on the presented system and provide comparison with related work. We also indicate the possible extension of the system and discuss the related issues.

2 The $\lambda P_{\hookrightarrow}$ calculus

In this section, the calculus $\lambda P_{\hookrightarrow}$ is defined, which is an extension of λP calculus with coercive subtyping along the line of Luo as introduced in [16] and [18].

The formal presentation begins with a grammar of *pre-terms* and judgement forms.

Definition 2.1 (Grammar of pre-terms). *Let x is from an enumerable set of variables, and α is from an enumerable set of type constants, then the pre-terms of the calculus may be constructed according to the following grammar:*

M, N	$::=$	$x \mid \lambda x{:}A.M \mid MN \mid \iota$	*terms of the calculus*
A, B	$::=$	$\alpha \mid \pi x{:}A.B \mid \Lambda x{:}A.B \mid AM$	*types of the calculus*
K	$::=$	$\star \mid \Pi x{:}A.K$	*kinds of the calculus*

As can be seen, the grammar can be divided to three syntactic categories:

- M denotes *terms* and can occur in three well-known forms—term variable, lambda abstraction or application. Term ι is a distinguished constant standing for identity

coercion. This distinction (from the obvious term $\lambda x{:}A.x$) is useful for deciding term equality during type derivation.

- A stands for *types*. In the simplest case a type can be represented by a *type constant* α.

  Next possible form, $\pi x{:}A.B$, denotes a π-type. π-types are known also as *dependent function types*. In various calculi without dependent types it is customary to denote a function type by $A \to B$. In our examples we use this notation, too: type $A \to B$ is an abbreviation for $\pi x{:}A.B$ where variable x does not occur in type term B.

  The third form, $\Lambda x{:}A.B$, is used to represent *type families*. This pre-term can be employed for construction of type functions taking a value and returning type (e.g. *list* is such type family and *list* n is the type of all lists of length n). Again, when variable x does not occur in type B, we abbreviate $\Lambda x{:}A.B$ by $A \Rightarrow B$.

  The last form is a *type application*.

- K denotes a *kind*. A pre-term of a *kind* can take either form of constant "$\star$" for the kind of *standard types*, or it can represent a more refined kind of value-dependent types (type families) in the form of $\Pi x{:}A.K$. To understand the difference and relation of *kinds* and *types* one can refer to fine clarification in [14].

2.1 Type System

The type system consists of *derivation rules* (typing, subtyping, kinding, and formation rules), composed of *judgements*. Judgements contain *contexts*, which is a (possibly empty) finite sequence of *declarations*.

Definition 2.2 (Context Declaration). *For every α being a type constant, K being a kind, A being a type, x being a variable and κ being a coercive function, a context declaration has one of the following forms:*

$\alpha : K$	*type α is in kind K*
$\kappa : \alpha \hookrightarrow A$	*type α is a subtype of A in kind $\star$ with coercive function κ.*
$x : A$	*variable x is assigned to type A.*

Definition 2.3 (Context). *Context is a finite sequence of context declarations.*

For the sake of clarity, any declaration may appear at most once in a given context. Thus, we rule out contexts like $\langle x{:}A, x{:}B\rangle$, or even $\langle x{:}A, x{:}A\rangle$.

For the presentation of the inference system we adapted Harper's equational formulation of LF [13]. Unlike the original system, we use only equational fragment and define a system of abbreviations that allows us to simulate the rest of the rules. The transformation to the Harper's presentation is easy. It can be shown that reflexivity judgements (e.g. $\Gamma \vdash K \Rightarrow \Gamma \vdash K = K$) are implicitly included in this presentation as e.g. $\Gamma \vdash K$ abbreviates $\Gamma \vdash K = K$.

Definition 2.4 (Judgement Forms). *Let* Γ *be a context,* K *be a kind,* A *and* B *be types,* κ *be a coercive function, and* M *be term, then there are four different judgement forms given as follows:*

$\Gamma \vdash K = K'$	K *and* K' *are equal kinds in context* Γ
$\Gamma \vdash A = A' : K$	A *and* A' *are equal types in kind K.*
$\Gamma \vdash \kappa : A \hookrightarrow B$	A *is a subtype of* B *in context* Γ *with* κ *being coercion function from* A *to* B.
$\Gamma \vdash M = N : A$	*terms* M *and* N *are equal in type* A.

In some cases, we may write $\Gamma \vdash M : A$ *instead of* $\Gamma \vdash M = M : A$ *and similarly* $\Gamma \vdash K$ *instead of* $\Gamma \vdash K = K$. *In judgements of form* $\Gamma \vdash \kappa : A \hookrightarrow B$ *we have designated* ι*-symbol that specifies a unique identity coercion.*

Rules for kind and context formation allow to create a well–defined context. They include the only axiom of the system, (F-EMPTY) rule, which simultaneously states that $\star$ is a well–formed kind and the empty context is a well–formed context.

Definition 2.5 (Formation Rules).

$$\frac{}{\langle\rangle \vdash \star}\ \text{F-EMPTY} \qquad \frac{\Gamma, x{:}A \vdash K = K' \qquad \Gamma \vdash A = A' : \star}{\Gamma \vdash \Pi x{:}A.K = \Pi x{:}A'.K'}\ \text{F-}\Pi$$

$$\frac{\Gamma \vdash A : \star}{\Gamma, x{:}A \vdash \star}\ \text{F-TERM} \qquad \frac{\Gamma \vdash K}{\Gamma, \alpha{:}K \vdash \star}\ \text{F-TYPE} \qquad \frac{\Gamma \vdash A : \star}{\Gamma, \kappa{:}\alpha \hookrightarrow A \vdash \star}\ \text{F-SUBT}$$

Similarly to [2], there are two possible ways of introducing type constants to the context. The rule (F-TYPE) allows one to insert to the context a new type constant that inhabits the given kind, e.g. $\Gamma, \mathit{seq} : \Pi x{:}\mathit{nat}.\star \vdash \star$. The other way is supposed to be employed if one needs to declare a subtype of an existing type. The rule (F-SUBT) enables declaring a new subtype and its accompanying coercion function, e.g. $\Gamma, n : \mathit{nat}, \kappa : \mathit{nlist} \hookrightarrow \mathit{seq}\ n \vdash \star$.

There is, however, the limitation imposed on the kind of types that can be subtype–related. Only types of $\star$-kind may be subtyped, which intuitively corresponds to the idea that coercions are applied to terms, hence only the inhabited types may be subtyped. As a side effect, this keeps coercion functions simply typed. This limitation is withdrawn in [15], where we admit also parameterized (lifted) coercions.

The n ext definition gives a collection of kinding rules. The purpose of these rules is to state the equality on types.

Definition 2.6 (Kinding Rules).

$$\text{K-VAR}\ \frac{\Gamma, \alpha{:}K, \Gamma' \vdash \star}{\Gamma, \alpha{:}K, \Gamma' \vdash \alpha : K} \qquad \text{K-CONV}\ \frac{\Gamma \vdash A_1 = A_2 : K_1 \qquad \Gamma \vdash K_1 = K_2}{\Gamma \vdash A_1 = A_2 : K_2}$$

$$\text{K-SYM}\ \frac{\Gamma \vdash A = B : K}{\Gamma \vdash B = A : K} \qquad \text{K-TRANS}\ \frac{\Gamma \vdash A = B : K \qquad \Gamma \vdash B = C : K}{\Gamma \vdash A = C : K}$$

$$\text{K-}\Lambda\ \frac{\Gamma, x{:}A_1 \vdash B_1 = B_2 : K \qquad \Gamma \vdash A_1 = A_2 : \star}{\Gamma \vdash \Lambda x{:}A_1.B_1 = \Lambda x{:}A_2.B_2 : \Pi x{:}A_1.K}$$

$$\text{K-APP}\ \frac{\Gamma \vdash A = A' : \Pi x{:}B'.K \qquad \Gamma \vdash M = M' : B \qquad \Gamma \vdash \kappa : B \hookrightarrow B'}{\Gamma \vdash A\ M = A'\ M' : K[x := M]}$$

$$\text{K-}\beta\ \frac{\Gamma, x{:}A \vdash B : K \qquad \Gamma \vdash M : A' \qquad \Gamma \vdash \kappa : A' \hookrightarrow A}{\Gamma \vdash (\Lambda x{:}A.B)\ M = B[x := M] : K[x := M]}$$

The kind conversion rule (K-CONV) is used to close kinding judgements under a conversion of well-formed kinds. As only possible general form of Π-type is $\Pi x_1{:}A_1 \ldots \Pi x_k{:}A_k[x_1, \ldots, x_{k-1}].\,\star$, it is sufficient to check equality of all argument types to assert that two kinds are equal. This principle is provided in rule (F-Π). The rule (K-Λ) is utilized for examining equality of arbitraty type families. The rule (K-APP) serves for the application of a type family to a well-typed term. By requiring a coercion function κ to exist, we allow here the type of the argument to be a subtype of an anticipated type in a type-family argument. The same applies also for (K-β). Rule (K-β) captures the notion of β–equality on the level of types. In this rule it is acceptable to supply a

term as a function argument, whose type is not equal to the expected argument type. Instead a subtype is admitted and the coercion function stands for the witness of this subsumption.

The following example demonstrates the application of (K-APP) rule. The example however does not consider subtyping of argument types.

Example 2.7 (Equivalence of type families). *Let us call "tall matrices" those whose width is smaller than their height,* $tm = \Lambda h{:}nat \Lambda w{:}less\ h.matrix\ h\ w$. *Let* Γ *be a context* $\langle nat : \star,\ tm : \Pi x{:}nat.less\ x \Rightarrow \star,\ m : nat,\ n : less\ m\rangle$. *Assume that we derive* $m' : nat$, $n' : less\ m$, *and also* $m = m'$ *and* $n = n'$ *(in the omitted part of the derivation below). Then the rule (*K-APP*) is used twice:*

$$\dfrac{\dfrac{\Gamma \vdash tm : \Pi x{:}nat.less\ x \Rightarrow \star \qquad \dfrac{\vdots}{\Gamma \vdash m = m' : nat}}{\Gamma \vdash tm\ m = tm\ m' : less\ m \Rightarrow \star}\ \text{K-APP} \qquad \dfrac{\vdots}{\Gamma \vdash n = n' : less\ m}}{\Gamma \vdash tm\ m\ n = tm\ m'\ n' : \star}\ \text{K-APP}$$

The subtyping judgements $\Gamma \vdash \kappa : A \hookrightarrow B$ consist of coercion term κ, which annotates the underlying subtyping relation. Although the coercion term is an ordinary lambda term, it can have in fact only one of the forms given by conclusions of the subtyping rules.

We use the following abbreviations:

$\kappa_1 \circ \kappa_2$ for $\lambda x{:}A.\kappa_1(\kappa_2\, x)$ where $\kappa_1 : A' \hookrightarrow A''$, $\kappa_2 : A \hookrightarrow A'$ are coercions
$(\circ\, \kappa)$ for $\lambda f{:}(\pi y{:}A'.B)\lambda x{:}A.f(\kappa\, x)$ where $\kappa : A \hookrightarrow A'$ is a coercion
$(\kappa \circ)$ for $\lambda f{:}(\pi y{:}A.B)\lambda x{:}A.\kappa(f\, x)$ where $\kappa : B \hookrightarrow B'$ is a coercion

Subtyping rules introduce coercion judgements for types of $\star$-kind. In these rules, the symbol ι may appear in the position of a coercion function, and indicates that two types are equal. The typing rule (T-ι) given later enables simplifying coercion terms containing identity coercions.

Definition 2.8 (Subtyping Rules).

$$\text{S-VAR}\ \frac{\Gamma, \kappa{:}\alpha{\hookrightarrow}A, \Gamma' \vdash \star}{\Gamma, \kappa{:}\alpha{\hookrightarrow}A, \Gamma' \vdash \kappa : \alpha \hookrightarrow A}$$

$$\text{S-}\pi1\ \frac{\Gamma \vdash \kappa : A \hookrightarrow A' \qquad \Gamma, x{:}A \vdash B : \star}{\Gamma \vdash (\circ\,\kappa) : (\pi x{:}A'.B) \hookrightarrow (\pi x{:}A.B)}$$

$$\text{S-}\pi2\ \frac{\Gamma, x{:}A \vdash \kappa : B \hookrightarrow B'}{\Gamma \vdash (\kappa\,\circ) : (\pi x{:}A.B) \hookrightarrow (\pi x{:}A.B')}$$

$$\text{S-REF}\ \frac{\Gamma \vdash A_1 = A_2 : \star}{\Gamma \vdash \iota : A_1 \hookrightarrow A_2} \qquad \text{S-TRANS}\ \frac{\Gamma \vdash \kappa_1 : A \hookrightarrow B \qquad \Gamma \vdash \kappa_2 : B \hookrightarrow C}{\Gamma \vdash \kappa_2 \circ \kappa_1 : A \hookrightarrow C}$$

It is important to realize that coercions are defined only between types of $\star$-kind. Rule (S-VAR) allows to use the declaration asserted previously into the context. Such assertion provides the basic coercion between two types, which are subtype-related. Rule (S-π1) provides us with a way to subtype function types by assuming contravariant typing of the function argument. Correspondingly, rule (S-π2) expresses that subtyped function types can be covariant in their result types. Rule (S-REF) is useful for introducing equality on types in subtyping judgements so that type equality and subtyping can be treated uniformly in typing judgements.

The next example demonstrates the use of rule (S-π1), which allows to subtype an argument type between (dependent) function types in a contravariant manner.

Example 2.9 (Contravariant subtyping). *Consider a context* $\Gamma \equiv \langle \mathit{nat} : \star,\ \mathit{list} : \mathit{nat} \Rightarrow \star,\ \kappa : \mathit{even} \hookrightarrow \mathit{nat} \rangle$.

$$\text{S-}\pi1\ \frac{\text{S-VAR}\ \dfrac{\vdots}{\Gamma \vdash \kappa : \mathit{even} \hookrightarrow \mathit{nat}} \qquad \text{K-APP}\ \dfrac{\vdots}{\Gamma, y{:}\mathit{even} \vdash \mathit{list}\ y : \star}}{\Gamma \vdash (\circ\,\kappa) : \pi x{:}\mathit{nat}.\mathit{list}\ x \hookrightarrow \pi y{:}\mathit{even}.\mathit{list}\ y}$$

Transitivity as subsumed by (S-TRANS) rule is necessary, if considering subtyping in several arguments of a function type. It immediatelly involves both (S-π1) and (S-π2) rules as demonstrated in the following example:

Example 2.10 (Multiple contravariant subtyping). *Let us consider context* $\Gamma \equiv \langle nat : \star,\ matrix : \Pi h{:}nat.\Pi w{:}nat.\star,\ \kappa : even \hookrightarrow nat \rangle$. *For the sake of space the following abbreviations for type expressions are used:*

$$\tau_{nn} \equiv \pi h{:}nat.\pi w{:}nat.matrix\ h\ w \to nat$$
$$\tau_{n} \equiv \pi w{:}nat.matrix\ h\ w \to nat$$
$$\tau_{e} \equiv \pi w{:}even.matrix\ h\ w \to nat$$
$$\tau_{en} \equiv \pi h{:}even.\pi w{:}nat.matrix\ h\ w \to nat$$
$$\tau_{ee} \equiv \pi h{:}even.\pi w{:}even.matrix\ h\ w \to nat$$

$$\dfrac{\dfrac{\Gamma \vdash \kappa : even \hookrightarrow nat}{\Gamma \vdash (\circ\,\kappa) : \tau_{nn} \hookrightarrow \tau_{en}}\ \text{S-}\pi1 \qquad \dfrac{\dfrac{\Gamma, h{:}even \vdash \kappa : even \hookrightarrow nat}{\Gamma, h{:}even \vdash (\circ\,\kappa) : \tau_{n} \hookrightarrow \tau_{e}}\ \text{S-}\pi1}{\Gamma \vdash ((\circ\kappa)\circ) : \tau_{en} \hookrightarrow \tau_{ee}}\ \text{S-}\pi2}{\Gamma \vdash ((\circ\kappa)\circ)\circ(\circ\kappa) : \tau_{nn} \hookrightarrow \tau_{ee}}\ \text{S-TRANS}$$

Finally, the last definition of the section introduces typing judgements. These rules assert equality on terms under the typing assumptions. This fragment depends on subtyping fragment as we allow the application of a function to the argument whose type is a subtype of the type expected by the function.

Definition 2.11 (Typing Rules).

$$\text{T-VAR}\ \frac{\Gamma, x{:}A, \Gamma' \vdash \star}{\Gamma, x{:}A, \Gamma' \vdash x : A} \qquad \text{T-CONV}\ \frac{\Gamma \vdash M_1 = M_2 : A_1 \qquad \Gamma \vdash A_1 = A_2 : \star}{\Gamma \vdash M_1 = M_2 : A_2}$$

$$\text{T-SYM}\ \frac{\Gamma \vdash M_1{=}M_2 : A}{\Gamma \vdash M_2{=}M_1 : A} \qquad \text{T-TRANS}\ \frac{\Gamma \vdash M_1{=}M_2 : A \qquad \Gamma \vdash M_2{=}M_3 : A}{\Gamma \vdash M_1{=}M_3 : A}$$

$$\text{T-}\iota\ \frac{\Gamma \vdash N_1{=}N_2 : A \qquad \Gamma \vdash \iota : A \hookrightarrow A}{\Gamma \vdash \iota\ N_1{=}N_2 : A}$$

$$\text{T-}\beta\ \frac{\Gamma, x{:}A \vdash M : B \qquad \Gamma \vdash N : A' \qquad \Gamma \vdash \kappa : A' \hookrightarrow A}{\Gamma \vdash (\lambda x{:}A.M)\ N{=}M[x := \kappa\ N] : B[x := N]}$$

$$\frac{\Gamma, x{:}A_1 \vdash M_1{=}M_2 : B \qquad \Gamma \vdash A_1{=}A_2 : \star}{\Gamma \vdash \lambda x{:}A_1.M_1{=}\lambda x{:}A_2.M_2 : \pi x{:}A_1.B}\ \text{T-}\lambda$$

$$\frac{\Gamma \vdash M_1{=}M_2 : \pi x{:}A.B \qquad \Gamma \vdash N_1{=}N_2 : A' \qquad \Gamma \vdash \kappa : A' \hookrightarrow A}{\Gamma \vdash M_1\,N_1{=}M_2\,N_2 : B[x := N_1]}\ \text{T-APP}$$

Rules (T-VAR), (T-SYM), (T-TRANS), and (T-λ) are standard. Rule (T-ι) defines equality under ι-contraction. It means that ι can be safely removed from the term as it does not represent a significant computational meaning. Rule (T-β) introduces β-reduction into equality judgements. There is, however, a significant difference to the usual β-reduction. A suitable coercion function is substituted with the argument during redex elimination by β-reduction. It means that the following two terms are, for instance, equal: $(\lambda x{:}\mathit{nat}.\mathit{twice}\ x)\ e = \mathit{twice}\ (\kappa\ e)$, if $\mathit{twice} : \mathit{nat} \to \mathit{even}$, $e : \mathit{even}$, and $\kappa : \mathit{even} \hookrightarrow \mathit{nat}$. Note that coercion is not inserted to types in (T-β) nor (T-APP) rules. The subtyping introduced via (T-APP) rule is shown in the following example.

Example 2.12 (Coercion in application). *Let* $\Gamma \equiv \langle \mathit{nat} : \star,\ \kappa : \mathit{even} \hookrightarrow \mathit{nat},\ \mathit{list} : \mathit{nat} \Rightarrow \star,\ \mathit{listMake} : \pi x{:}\mathit{nat}.\mathit{list}\ x,\ e : \mathit{even}\rangle$*, then:*

$$\frac{\Gamma \vdash \mathit{listMake} : \pi x{:}\mathit{nat}.\mathit{list}\ x \qquad \Gamma \vdash e : \mathit{even} \qquad \Gamma \vdash \kappa : \mathit{even} \hookrightarrow \mathit{nat} : \star}{\Gamma \vdash \mathit{listMake}\ e : \mathit{list}\ e}\ \text{T-APP}$$

One may attempt to define a subtype for an arbitrary π-type. Such directly defined subtype, however, does not express enough information and its use is very limited. The present type system even does not offer a way of typing expressions involving such subtypes in a supertype context. The following example demonstrates an attempt to type such application.

Example 2.13 (Direct subtypes of π-types). *Let* $\Gamma \equiv \langle \kappa : \alpha \hookrightarrow \pi x{:}A.B,\ f : \alpha,\ a : A\rangle$*. If we added the general subsumption rule, the following would be possible:*

$$\frac{\dfrac{\Gamma \vdash f : \alpha \qquad \Gamma \vdash \kappa : \alpha \hookrightarrow \pi x{:}A.B}{\Gamma \vdash f : \pi x{:}A.B}\ \text{SUBSUM} \qquad \dfrac{\vdots}{\Gamma \vdash a : A}}{\Gamma \vdash f\ a : B}$$

We do not have general subsumption rule (SUBSUM) in our system. Having it, we could deduce that $f\ a : B'$ for some $B \leq B'$. As we cannot bind such type it is not possible to declare any canonical object of type B'. From this viewpoint, it seems to be undesirable to introduce direct subtypes of π-types.

2.2 Reduction

The reduction relation on terms and types can be defined by means of the typing rules (T-ι) and (T-β) (reduction on terms), and the kinding rule (K-β) (reduction on types). This means that reduction requires typing to work and the only correct notion of reduction is *typed reduction*. We give an operational semantics to the calculus that captures the notion of typed reduction. The method used here follows the Goguen's approach [12]. This proceeds by introducing a system, denoted $\lambda P^{R}_{\hookrightarrow}$, for typed operational semantics, which can be seen as a type theory in which computation is the central notion instead of logical inference. To distinguish this system from $\lambda P_{\hookrightarrow}$ calculus, we will systematically write $\vdash^{R}$ in judgements.

Definition 2.14 (Judgements Forms of $\lambda P^{R}_{\hookrightarrow}$).

$\Gamma \vdash^{R} M \rightsquigarrow^{nf} P : A$	M *has canonical form* P *which is a canonical element of type* A *in context* Γ
$\Gamma \vdash^{R} M \rightsquigarrow^{wh} N : A$	M *weak head reduces to* N *of type* A *in context* Γ

Remember that the term is in weak head normal form if its outermost term is not a redex. Then to obtain a normal form for such term it is sufficient to perform only internal reductions.

The typed operational semantics is defined by the rules of inference in Fig. 1. The rules reflect the intended typed reductions of the calculus. They can be also viewed as equality rules in $\lambda P_{\hookrightarrow}$ without symmetry property. The derivations in $\lambda P^{R}_{\hookrightarrow}$ relate normal forms that can be obtained by applications of weak head reductions.

3 Properties of $\lambda P_{\hookrightarrow}$

Our calculus has standard useful properties like strong normalization, or Church-Rosser property. This is because the only essential difference from $\lambda P_{\leq}$ is in explicit coercions supplied in judgements, and this additional information does not spoil the properties of $\lambda P_{\leq}$. We also need to deal with equational judgements which allows us

R-T-VAR
$$\frac{}{\Gamma \vdash^{R} x \rightsquigarrow^{nf} x : A}$$

R-λ
$$\frac{\Gamma, x{:}A \vdash^{R} M_1 \rightsquigarrow^{nf} M_2 : B}{\Gamma \vdash^{R} \lambda x{:}A.M_1 \rightsquigarrow^{nf} \lambda x{:}A.M_2 : \pi x{:}A.B}$$

R-K-VAR
$$\frac{}{\Gamma \vdash^{R} \alpha \rightsquigarrow^{nf} \alpha : A}$$

R-Λ
$$\frac{\Gamma, x{:}A \vdash^{R} B_1 \rightsquigarrow^{nf} B_2 : K}{\Gamma \vdash^{R} \Lambda x{:}A.B_1 \rightsquigarrow^{nf} \Lambda x{:}A.B_2 : \Pi x{:}A.K}$$

R-T-APP
$$\frac{\Gamma \vdash^{R} M_1 \rightsquigarrow^{nf} M_2 : \pi x{:}A.B \qquad \Gamma \vdash^{R} N_1 \rightsquigarrow^{nf} N_2 : A' \qquad \Gamma \vdash^{R} \kappa : A' \hookrightarrow A}{\Gamma \vdash^{R} M_1\, N_1 \rightsquigarrow^{nf} M_2\, N_2 : B[x := N_1]}$$

R-K-APP
$$\frac{\Gamma \vdash^{R} B_1 \rightsquigarrow^{nf} B_2 : \Pi x{:}A.K \qquad \Gamma \vdash^{R} N_1 \rightsquigarrow^{nf} N_2 : A' \qquad \Gamma \vdash^{R} \kappa : A' \hookrightarrow A}{\Gamma \vdash^{R} B_1\, N_1 \rightsquigarrow^{nf} B_2\, N_2 : K[x := N_1]}$$

R-T-WH
$$\frac{\Gamma \vdash^{R} M_1 \rightsquigarrow^{wh} M_2 : A \qquad \Gamma \vdash^{R} M_2 \rightsquigarrow^{nf} M_3 : A}{\Gamma \vdash^{R} M_1 \rightsquigarrow^{nf} M_3 : A}$$

R-K-WH
$$\frac{\Gamma \vdash^{R} A_1 \rightsquigarrow^{wh} A_2 : K \qquad \Gamma \vdash^{R} A_2 \rightsquigarrow^{nf} A_3 : K}{\Gamma \vdash^{R} A_1 \rightsquigarrow^{nf} A_3 : K}$$

W-T-β
$$\frac{\Gamma, x{:}A \vdash^{R} M : B \qquad \Gamma \vdash^{R} N : A' \qquad \Gamma \vdash^{R} \kappa : A' \hookrightarrow A}{\Gamma \vdash^{R} (\lambda x{:}A.M)\, N \rightsquigarrow^{wh} M[x := \kappa\, N] : B[x := N]}$$

W-T-APP
$$\frac{\Gamma \vdash^{R} M_1 \rightsquigarrow^{wh} M_2 : \pi x{:}A.B \qquad \Gamma \vdash^{R} N : A' \qquad \Gamma \vdash^{R} \kappa : A' \hookrightarrow A}{\Gamma \vdash^{R} M_1\, N \rightsquigarrow^{wh} M_2\, N : B[x := N]}$$

W-T-ι
$$\frac{\Gamma \vdash^{R} N_1 \rightsquigarrow^{wh} N_2 : A' \qquad \Gamma \vdash^{R} \iota : A' \hookrightarrow A}{\Gamma \vdash^{R} \iota\, N_1 \rightsquigarrow^{wh} N_2 : A}$$

W-K-β
$$\frac{\Gamma, x{:}A \vdash^{R} B : K \qquad \Gamma \vdash^{R} N : A' \qquad \Gamma \vdash^{R} \kappa : A' \hookrightarrow A}{\Gamma \vdash^{R} (\Lambda x{:}A.B)\, N \rightsquigarrow^{wh} B[x := N] : K[x := N]}$$

W-K-APP
$$\frac{\Gamma \vdash^{R} B_1 \rightsquigarrow^{wh} B_2 : \Pi x{:}A.K \qquad \Gamma \vdash^{R} N : A' \qquad \Gamma \vdash^{R} \kappa : A' \hookrightarrow A}{\Gamma \vdash^{R} B_1\, N \rightsquigarrow^{wh} B_2\, N : K[x := N]}$$

Figure 1: Inference Rules of Typed Operational Semantics

to consider typed reductions on terms and types. Typing information in reductions is necessary to supply right coercions in expressions during the evaluation.

First, we show the correspondence of defined typed operational semantics with respect to the equational judgements of the calculus.

Proposition 3.1 (Soundness of Typed Operational Semantics). *Let* M, N, *and* P *be well-formed terms and* A, B, *and* C *be well-formed types, then:*

- $\Gamma \vdash M = N : A$ *then* $\Gamma \vdash M \rightsquigarrow^{nf} P : A$ *and* $\Gamma \vdash N \rightsquigarrow^{nf} P : A$.
- $\Gamma \vdash A = B : K$ *then* $\Gamma \vdash A \rightsquigarrow^{nf} C : K$ *and* $\Gamma \vdash B \rightsquigarrow^{nf} C : K$.

PROOF. The proof is done by simultaneous induction on derivations in $\lambda P_{\leq}$. Similarly to presentation in [12], it is required to deal with normalization through introducing semantics objects and term interpretation accompanied with stating several supporting lemmas.

Because of the previous result, it is possible to show strong normalization, subject reduction and confluence in system of typed operational semantics, with the conclusion that $\lambda P_{\hookrightarrow}$ shares these properties, too.

Proposition 3.2 (Strong Normalization). *If* $\Gamma \vdash M : A$ *then* M *is strongly normalizing.*

The important property is the decidability of the calculus. The idea of the proof is based on the observation that subtyping fragment has only little dependency on other rules if it works only with canonical representation of terms and types. This allows us to remove rule (S-REF) as it only introduces ι coercion, which has no computational meaning. Thus we can consider only coercions that do not contain ι in canonical representation.

Proposition 3.3 (Decidability). *The derivability of a given well-formed judgement* $\Gamma \vdash J$ *is decidable.*

PROOF. Let us split the calculus $\lambda P_{\hookrightarrow}$ in two parts (partial calculi).

First we leave out all rules involving subtyping ("S-rules") and replace rules (K-β), (K-APP), (T-β), (T-APP), by standard rules of λP calculus—i.e. disregarding coercion judgements and treating coercion as identity. What we get is dependently-typed lambda calculus with equality relation. The standard de-pendently-typed lambda calculus λP (without equality relation) is decidable (see [3]).

Equality of terms, and hence also equality of types and kinds, is decidable due to strong normalization and Church-Rosser property of reduction. Thus judgements of forms $\Gamma \vdash M = N : A$, $\Gamma \vdash A = B : K$, are decidable. The typing and kinding judgements of forms $\Gamma \vdash M : A$, $\Gamma \vdash A : K$ respectively, are derived by the rules that do not have equality judgements in premises. (Strictly speaking, they do have, but a judgement $\Gamma \vdash M : A$, being an abbreviation of $\Gamma \vdash M = M : A$, is derived only from judgements of the form $\Gamma \vdash N = N : B$, where the equivalence $N = N$ is syntactic identity.)

Second, take a fragment involving only formation rules and subtyping rules without rule (S-REF). We get an instance of simply typed lambda calculus—there is no dependence in the types of the coercions. Thus this fragment of $\lambda P_{\hookrightarrow}$ is decidable.

Third, we have to show that combining these two fragments in $\lambda P_{\hookrightarrow}$ does not harm the decidability. The only problem, that could potentially endanger decidability, is a circular dependence of derivation rules in both fragment calculi. Notice that there are several typing rules that have subtyping judgements among their premises, and there is one subtyping rule (see (S-REF)), which has typing judgement in its conclusion. However, the latter can be eliminated. The rule (S-REF) produces only subtyping judgements with identity coercions ι, that play rather redundant role in coercion terms—ι coercions can be safely erased from the coercion terms.

4 Conclusions

We formalize a calculus of coercive subtyping, which allows integration of dependent types and a restricted form of subtyping in a uniform type theoretic system. The coercion functions are necessary for transformation of terms that inhabit types in subtyping relation. The theory makes possible to define basic coercions that come with new subtype definitions and to build compositions of coercions upon this system. The subtyping fragment of a system is mostly independent from the original system. Therefore it is easier to analyze the properties of the calculus. Moreover, the fragment itself can be seen as a simply typed calculus defined on coercion functions.

There are several related works to ours. We were initially motivated by calculus $\lambda P_{\leq}$ by Aspinall and Compagnoni [2]. In their calculus, the subtyping is added to λP by defining subtyping relation that includes term overloading. It is not always possible to allow term overloading in type theory as it was shown in case of inductive types [11]. Instead, we attempted to base the system on coercive subtyping, which was shown to

be a more powerful tool in systems of type theories and can be extended easier with constructs such as inductive types.

Considering coercion $\kappa : A \hookrightarrow B$ to be a unique mapping from terms of type A to terms of type B, we feel that coercions should be bound to $\star$-types as only these types are inhabited in λP. In this paper, we therefore restrict the coercion only to these types, which significantly reduces the set of typeable expressions compared to $\lambda P_{\leq}$. Although for some fixed n and $\mathit{nlist} : \star$ we may derive $\Gamma \vdash \kappa : \mathit{nlist} \hookrightarrow \mathit{seq}\ n$, it turns out that it is not possible to have a generalized version:

$$\pi x{:}\mathit{nat}.\mathit{list}\ x \hookrightarrow \pi x{:}\mathit{nat}.\mathit{seq}\ x$$

It is because the way the subtypes are introduced in the context under the restriction that they may be only $\star$-types. This prevents us from using (F-SUBT) rule for introducing $\kappa : \mathit{list} \hookrightarrow \mathit{seq}$ as these are types of kind $\mathit{nat} \Rightarrow \star$. Allowing this we would arrive to a system of parameterized coercions (coercion schemata):

$$\mathit{seq} : \mathit{nat} \Rightarrow \star,\ \kappa[x{:}\mathit{nat}] : \mathit{list} \hookrightarrow \mathit{seq}$$

Requiring that coercions are definitional terms of the calculus, we should write $\kappa[x{:}\mathit{nat}] \equiv \Lambda x{:}\mathit{nat}.\kappa_x$. This approach lifts the coercions from $\star$-kind to Π-kinds and we need to deal with dependent coercions. The study of such extension of the system is considered as the future work.

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[2] D. Aspinall and A. Compagnoni. Subtyping dependent types (summary. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 86–97. IEEE Computer Society Press, 1996.

[3] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures)*, volume 2. Clarendon Press, 1992.

[4] G. Barthe. Implicit coercions in type systems. In *In Selected Papers from the International Workshop TYPES 1995*, pages 1–15. Springer-Verlag, 1996.

[5] G. Betarte. *Dependent Record Types and Formal Abstract Reasoning*. PhD thesis, Chalmers University of Technology, 1998.

[6] G. Betarte. Dependent record types, subtyping and proof reutilization. In *Subtyping, inheritance and modular development of proofs*, 1998.

[7] L. Cardelli. Typechecking dependent types and subtypes. In *Proceedings of the Workshop on Foundations of Logic and Functional Programming*, pages 45–57, London, UK, 1988. Springer-Verlag.

[8] G. Chen. Subtyping calculus of construction. In *Mathematical Foundations of Computer Science*, pages 189–198. Springer-Verlang, 1997.

[9] G. Chen. Dependent type system with subtyping. *Journal of Computer Science and Technology*, 13(6), 1998.

[10] G. Chen. Coercive subtyping for the calculus of constructions. *SIGPLAN Not.*, 38(1):150–159, 2003.

[11] T. Coquand. Pattern matching with dependent types. In *In Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, 1992.

[12] H. Goguen. *A typed operational semantics for type theory*. PhD thesis, University of Edinburgh, 1994.

[13] R. Harper. An equational formulation of lf. Technical Report ECS-LFCS-88-67, University of Edingurgh, 1988.

[14] S. Jones and E. Meijer. Henk: a typed intermediate language. In *Proceedings of ACM SIGPLAN Workshop on Types in Compilation*. ACM Press, June 1997.

[15] M. Kollár, L. Škarvada, O. Peterka, and O. Ryšavý. A calculus of coercive subtyping. In *Draft Proceedings of the 21st International Symposium on Implementation and Application of Functional Languages*, SHU-TR-CS-2009-09-01, pages 182–192, 2009.

[16] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[17] Z. Luo. Coercions in a polymorphic type system. *Mathematical. Structures in Comp. Sci.*, 18(4):729–751, 2008.

[18] Z. Luo and S. Soloviev. Dependent coercions. In *Proceedings of 8th conference on Category Theory and Computer Science (CTCS'99)*, 1999.

[19] F. Pfenning. Refinement types for logical frameworks. In *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, 1993.

[20] J. Zwanenburg. Pure type systems with subtyping. In *TLCA '99: Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, pages 381–396, London, UK, 1999. Springer-Verlag.

A Syntax

Definition A.1 (Grammar of pre-terms). *Let x is from an enumerable set of variables, and α is from an enumerable set of type constants, then the pre-terms of the calculus may be constructed according to the following grammar:*

M, N	$::=$	$x \mid \lambda x{:}A.M \mid MN \mid \iota$	*terms of the calculus*
A, B	$::=$	$\alpha \mid \pi x{:}A.B \mid \Lambda x{:}A.B \mid AM$	*types of the calculus*
K	$::=$	$\star \mid \Pi x{:}A.K$	*kinds of the calculus*

B Formation and Equality Rules

Definition B.1 (Formation Judgements). *The formation judgements are as follows:*

$\Gamma \vdash K$	K *is a valid kind*
$\Gamma \vdash A : K$	A *is a family of kind* K
$\Gamma \vdash M : A$	M *is an object of type* A.

Definition B.2 (Equality Judgements). *The equality judgements are as follows:*

$\Gamma \vdash K_1 = K_2$	K_1 *and* K_2 *are equal kinds*
$\Gamma \vdash A_1 = A_2 : K$	A_1 *and* A_2 *are equal families of kind* K
$\Gamma \vdash M_1 = M_2 : A$	M_1 *and* M_2 *are equal objects of type* A.