Foundations for the Implementation of Higher-Order Subtyping

Karl Crary†
Department of Computer Science
Cornell University

Abstract

We show how to implement a calculus with higher-order subtyping and subkinding by replacing uses of implicit subsumption with explicit coercions. To ensure this can be done, a polymorphic function is adjusted to take, as an additional argument, a proof that its type constructor argument has the desired kind. Such a proof is extracted from the derivation of a kinding judgement and may in turn require proof coercions, which are extracted from subkinding judgements. This technique is formalized as a type-directed translation from a calculus of higher-order subtyping to a subtyping-free calculus. This translation generalizes an existing result for second-order subtyping calculi (such as $F_{\leq}$).

We also discuss two interpretations of subtyping, one that views it as type inclusion and another that views it as the existence of a well-behaved coercion, and we show, by a type-theoretic construction, that our translation is the minimum consequence of shifting from the inclusion interpretation to the coercion-existence interpretation. This construction shows that the translation is the natural one, and it also provides a framework for extending the translation to richer type systems. Finally, we show how the two interpretations can be reconciled in a common semantics. It is then easy to show the coherence of the translation relative to that semantics.

1 Introduction

A subtyping calculus is characterized by a subtyping relation between types and the subsumption rule: if τ_1 is a subtype of τ_2 (written $\tau_1 \leq \tau_2$) and a term e has type τ_1, then e can be used as a member of type τ_2. This rule is at the heart of a subtyping calculus, but it poses a problem for an implementer of the calculus because if τ_1 and τ_2 have different machine representations, the compiler must ensure that a computation expecting an argument of type τ_2 is given that object in the machine representation of τ_2, not that of τ_1.

†Support for this research was provided by the Office of Naval Research through grant N00014-92-J-1764 and by the National Science Foundation through grant CCR-9244739.

Appeared in the 1997 International Conference on Functional Programming, Amsterdam, The Netherlands, June 1997.

For example, a subtyping system might include the axiom $int \leq real$, reflecting the mathematical fact that every integer is also a real number. In such a system it would be legal to write the code `sqrt(2)` where the square root function, `sqrt`, is a function on reals. However, integers and reals (or, more properly, floats) have very different machine representations, so the compiler must coerce `2` to be a float before passing it to `sqrt`.

One approach would be to tag all objects with their types and perform the coercions at run-time. However, this is likely unacceptable due to the performance penalty it would entail. To have acceptable performance, the compiler must determine the proper coercions statically, at compile-time.

This can easily be done in an (explicitly typed) programming language with first-class functions, records, and even parametric polymorphism, but becomes complicated with the introduction of bounded quantification [12]. Consider the bounded polymorphic function $\Lambda\alpha{\leq}\tau.\, \lambda x{:}\alpha.\, f(x)$ where f has type $\tau \to \tau'$. With α unknown, it is not possible to determine the coercion for x from type α to type τ. Since $\alpha \leq \tau$, we know there exists a coercion from α to τ, but we have no idea what it is. (The reader familiar with constructive logic may notice a parallel between this difficulty and the problem of extracting programs from non-constructive proofs; we will see later that indeed the solution results from making kinding proofs constructive.)

Breazu-Tannen *et al.* [4], hereafter referred to as BCGS, showed how to solve this problem in the second-order case for languages such as Fun [12] and $F_{\leq}$ [11, 10]. The basic idea is for bounded polymorphic functions to take the necessary coercion as an additional argument. In this paper we show how to extend this technique to languages with higher-order type constructors and a subkinding relation, such as Quest [9], $F^{\omega}_{\leq}$ [8, 28] and λ^{K} [16]. Instead of taking a coercion as an extra argument, as in BCGS, polymorphic functions take a higher-order kinding proof. In the second-order case, this kinding proof turns out to be a simple coercion, so the higher-order solution proves to be a generalization of the second-order one. We formalize the technique by giving a translation of programs from a source calculus into a subsumption-free target calculus. This translation is used in the compiler for the KML programming language [15].

We also give a type-theoretic construction that explains the translation as a construction of the source calculus that maintains constructivity while changing the interpretation of subtyping from one based on type inclusion to one based on the existence of semantics-preserving coercions. This construction gives a deeper understanding of the translation than simply as a syntactic device, it shows that our translation is in some sense the natural one, and it also gives a framework for extending the translation to richer type systems.

The two interpretations of subtyping seem to be disparate, but we show that they can be reconciled in a common semantics by quotienting the equalities of types modulo coercions. In such a semantics it is easy to show the translation's coherence. Aside from this coherence result, the common semantics is also useful for the formal definition of programming languages: While a implementation-oriented definition would specify the insertion of coercions into program code, a type-theoretic definition would be cleaner if those coercions were left out. With a common semantics available for each interpretation of subtyping, a language designer can employ both styles of definition and show a formal equivalence between them.

This paper is organized as follows: In Section 2 we briefly present the source and target calculi of our translation and in Section 3 we give the translation itself. These sections are a fairly straightforward extension of the BCGS result, and the emphasis there is on exposition for the reader less familiar with the technique. The remaining sections contain the more novel contributions of the paper: In Section 4 we show the type-theoretic construction motivating the translation; this section assumes some familiarity with constructive type theory and the propositions-as-types principle. In Section 5 we semantically reconcile the two subtyping interpretations and use this semantics to give simple conditions under which the translation is coherent. Related work is discussed in Section 6 and brief concluding remarks appear in Section 7.

2 Source and Target Calculi

The Source We begin by giving the definition of our source calculus. In the literature, this calculus is closest to Quest [9], $F^{\omega}_{\leq}$ [8] and λ^{K} [16]. The syntax rules appear in Figure 1. The source consists of three syntactic classes: kinds, type constructors (often referred to briefly as "constructors") and terms. A fourth syntactic class, contexts, is used to assign meaning to free variables in the typing rules. The term class contains variables and the usual introduction and elimination forms for functions, polymorphic functions and records. The type constructor class contains the types of these terms (arrow, quantified and record types), a top type, a bottom type "*void*," and a simply typed lambda calculus (with pairs) over those types. Thus, the class of type constructors contains types and higher-order type constructors with which types can be built.

The kind class, which gives the "types" of type constructors, is the most novel. It contains the power kind [7] and dependent product and sum kinds. When τ is a type, the power kind $\mathcal{P}(\tau)$ contains all types that are subtypes of τ. Since every type is a subtype of *Top*, the kind of all types, *Type*, is defined as $\mathcal{P}(\mathit{Top})$. The dependent product kind, $\Pi\alpha{:}\kappa_1.\kappa_2$, is the kind of abstractions from kind κ_1 to kind κ_2 where α stands for the argument and may appear free in κ_2. Likewise, the dependent sum kind, $\Sigma\alpha{:}\kappa_1.\kappa_2$, contains pairs of a κ_1 and a κ_2, where α stands for the left member and may appear free in κ_2.

The typing rules for the source appear in Appendix A. The system contains judgements for typing, kinding, subkinding (denoted $\preceq$) and well-formedness of contexts. By using the power kind, the kinding judgement does double duty to specify the subtyping relation. Since constructors appear within kinds, not all kinds are well-formed, so some judgement must specify well-formedness of kinds; reflexive subkinding judgements serve this purpose.

kinds	κ ::=	$\mathcal{P}(c) \mid \Pi\alpha{:}\kappa_1.\kappa_2 \mid \Sigma\alpha{:}\kappa_1.\kappa_2$
constructors	c ::=	$\alpha \mid \lambda\alpha{:}\kappa.c \mid c_1[c_2] \mid \langle c_1, c_2\rangle \mid \pi_1(c) \mid \pi_2(c) \mid c_1 \rightarrow c_2 \mid \forall\alpha{:}\kappa.c \mid \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \mid \mathit{Top} \mid \mathit{void}$
terms	e ::=	$x \mid \lambda x{:}c.e \mid e_1 e_2 \mid \Lambda\alpha{:}\kappa.e \mid e[c] \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid \pi_\ell(e)$
contexts	Γ ::=	$\bullet \mid \Gamma[\alpha : \kappa] \mid \Gamma[x : c]$

Figure 1: Source Calculus Syntax Rules

kinds	κ ::=	$\mathit{Type} \mid \kappa_1 \rightarrow \kappa_2 \mid \kappa_1 \times \kappa_2$
constructors	c ::=	$\alpha \mid \lambda\alpha{:}\kappa.c \mid c_1[c_2] \mid \langle c_1, c_2\rangle \mid \pi_1(c) \mid \pi_2(c) \mid c_1 \rightarrow c_2 \mid \forall\alpha{:}\kappa.c \mid \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \mid \mathit{unit} \mid \mathit{void}$
terms	e ::=	$x \mid \lambda x{:}c.e \mid e_1 e_2 \mid \Lambda\alpha{:}\kappa.e \mid e[c] \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid \pi_\ell(e) \mid \diamond \mid \mathit{any}_c(e)$
contexts	Γ ::=	$\bullet \mid \Gamma[\alpha : \kappa] \mid \Gamma[x : c]$

Figure 2: Target Calculus Syntax Rules

The Target The target calculus is essentially Girard's F_ω [19, 20], augmented with a few additional constructs. The syntax rules appear in Figure 2. There are just a few differences between the source and target calculi, all stemming from the absence of subtyping: At the kind level the kind *Type* is now primitive, replacing the power kind. Without power kinds, constructors can no longer appear within kinds, rendering dependent kinds unnecessary, so those are replaced by non-dependent ones. At the type constructor level the type *Top* is removed and replaced by the trivial type *unit*; also the *void* type is retained as an empty type but is no longer considered a bottom type. At the term level, since subsumption can no longer be used to view an element of *void* as an element of any type, an explicit elimination form for *void* is added; $\mathit{any}_\tau(e)$ has type τ whenever e has type *void* and τ is a type. Finally, an introduction form $\diamond$ is included for *unit*.

The typing rules for the target appear in Appendix B and are similar to those for the source, except that subsumption is omitted, as are all rules for subtyping and subkinding.

3 Eliminating Subsumption

Before launching into the translation itself, we first motivate it with a few (contrived) examples. Consider the bounded polymorphic function

$$\Lambda\alpha{:}\mathcal{P}(\mathit{intlist}).\, \lambda x{:}\alpha.\, \mathit{sum}\, x$$

where *sum* has type $\mathit{intlist} \rightarrow \mathit{int}$. This can be translated, as in BCGS, as:

$$\Lambda\alpha{:}\mathit{Type}.\, \lambda\hat{\alpha}{:}\alpha \rightarrow \mathit{intlist}.\, \lambda x{:}\alpha.\, \mathit{sum}(\hat{\alpha}\, x)$$

Implicit in this translation is the interpretation of subtyping as coercion-existence: that $\alpha \leq \mathit{intlist}$ exactly when there is a coercion from α to *intlist*. Thus, any coercion $\alpha \rightarrow \mathit{intlist}$ is a proof that α has kind $\mathcal{P}(\mathit{intlist})$. We translate a polymorphic function to one taking a type constructor argument and a proof that the argument has the proper kind.

Now consider the higher-order bounded polymorphic function:

$$\Lambda\gamma{:}(\Pi\alpha{:}\mathcal{P}(\mathit{int}).\,\mathcal{P}(\mathit{list}[\alpha])).\ \lambda x{:}\gamma[\mathit{int}].\ \mathit{sum}\ x$$

When a type constructor γ has kind $\Pi\alpha{:}\mathcal{P}(\mathit{int}).\mathcal{P}(\mathit{list}[\alpha])$, it means that for any α, if α has kind $\mathcal{P}(\mathit{int})$ then γ applied to α has kind $\mathcal{P}(\mathit{list}[\alpha])$. Under the coercion-existence subtyping interpretation, this in turn means that for any α, if there is a coercion $\alpha \rightarrow \mathit{int}$, then there is a coercion $\gamma[\alpha] \rightarrow \mathit{list}[\alpha]$. Thus, proofs that γ has this kind have the polymorphic type $\forall\alpha{:}\mathit{Type}.\,(\alpha \rightarrow \mathit{int}) \rightarrow \gamma[\alpha] \rightarrow \mathit{list}[\alpha]$. That proof, call it $\hat{\gamma}$, is used to translate x from type $\gamma[\mathit{int}]$ to type $\mathit{list}[\mathit{int}]$ by applying it first to *int* and then to the proof that *int* has kind $\mathcal{P}(\mathit{int})$ (which is the identity coercion on $\mathit{int} \rightarrow \mathit{int}$), resulting in $\hat{\gamma}[\mathit{int}]\,(\lambda z{:}\mathit{int}.z)\,x$. Hence the term is translated:

$$\begin{array}{l}\Lambda\gamma{:}\mathit{Type} \rightarrow \mathit{Type}.\\ \quad \lambda\hat{\gamma}{:}(\forall\alpha{:}\mathit{Type}.(\alpha \rightarrow \mathit{int}) \rightarrow \gamma[\alpha] \rightarrow \mathit{list}[\alpha]).\\ \qquad \lambda x{:}\gamma[\mathit{int}].\ \mathit{sum}\ (\hat{\gamma}[\mathit{int}]\ (\lambda z{:}\mathit{int}.z)\ x)\end{array}$$

In each of these examples, the coercion used in subsumption was extracted directly from the proof that some variable has its given kind. In general, of course, a coercion may involve the proofs of more than one variable (or none at all) and some additional computation. This construction of a coercion is driven by the derivation of the subtyping judgement.

The process of constructing a coercion can be understood as follows: In the subsumption rule we are given that $\tau_1 \leq \tau_2$. This is shorthand for $\tau_1 : \mathcal{P}(\tau_2)$, and, under the coercion-existence interpretation, this indicates the existence of a coercion from τ_1 to τ_2. A proof of this kinding relationship would be such a coercion. The translation extracts such a proof from the derivation of $\tau_1 \leq \tau_2$ in a manner reminiscent of the extraction of programs from constructive proofs [3]. The additional proof variables shown above (*e.g.,* $\hat{\alpha}$, $\hat{\gamma}$) ensure that proofs are available at the leaves of the derivation.

As an aside, we can see here the potential for a computational inefficiency. Consider the polymorphic type $\forall\alpha{:}\mathcal{P}(\mathit{Top}).\,\mathit{list}[\alpha] \rightarrow \mathit{list}[\alpha]$, which might be assigned to a list reversal function. Proofs that α has kind $\mathcal{P}(\mathit{Top})$ are of the type $\alpha \rightarrow \mathit{unit}$ (we will translate *Top* as *unit*), so this type would be translated $\forall\alpha{:}\mathit{Type}.\,(\alpha \rightarrow \mathit{unit}) \rightarrow \mathit{list}[\alpha] \rightarrow \mathit{list}[\alpha]$. However, all coercions $\alpha \rightarrow \mathit{unit}$ are equal, since *unit* has only one (convergent) element, so the kinding proof need not, in practice, be taken as an argument. A compiler based upon this technique would suppress trivial proofs, as does the KML compiler. Nevertheless, for the sake of simplicity, we will not suppress trivial proofs in this paper.

3.1 The Type-Directed Translation

We are now ready to begin the formal details of the translation. A kind is translated by converting power kinds to *Type* and erasing dependency information, as shown in Figure 3 (where the translation of κ is denoted $\overline{\kappa}$).

$$\begin{array}{rcl}\overline{\Pi\alpha{:}\kappa_1.\kappa_2} & = & \overline{\kappa_1} \rightarrow \overline{\kappa_2}\\ \overline{\Sigma\alpha{:}\kappa_1.\kappa_2} & = & \overline{\kappa_1} \times \overline{\kappa_2}\\ \overline{\mathcal{P}(c)} & = & \mathit{Type}\end{array}$$

Figure 3: Kind Translation

$$\begin{array}{rcl}|\alpha| & = & \alpha\\ |\lambda\alpha{:}\kappa.c| & = & \lambda\alpha{:}\overline{\kappa}.|c|\\ |c_1[c_2]| & = & |c_1|\,[|c_2|]\\ |\langle c_1, c_2\rangle| & = & \langle |c_1|, |c_2|\rangle\\ |\pi_i(c)| & = & \pi_i(|c|)\\ |c_1 \rightarrow c_2| & = & |c_1| \rightarrow |c_2|\\ |\forall\alpha{:}\kappa.c| & = & \forall\alpha{:}\overline{\kappa}.\,\varphi(\alpha,\kappa) \rightarrow |c|\\ |\{\ell_1 : c_1, \ldots, \ell_n : c_n\}| & = & \{\ell_1 : |c_1|, \ldots, \ell_n : |c_n|\}\\ |\mathit{Top}| & = & \mathit{unit}\\ |\mathit{void}| & = & \mathit{void}\\ & & \\ \varphi(c, \mathcal{P}(c')) & = & c \rightarrow |c'|\\ \varphi(c, \Pi\alpha{:}\kappa_1.\kappa_2) & = & \forall\alpha{:}\overline{\kappa_1}.\,\varphi(\alpha,\kappa_1) \rightarrow \varphi(c[\alpha],\kappa_2)\\ \varphi(c, \Sigma\alpha{:}\kappa_1.\kappa_2) & = & \varphi(\pi_1(c),\kappa_1) \times\\ & & \quad \varphi(\pi_2(c),\kappa_2)[\pi_1(c)/\alpha]\end{array}$$

Figure 4: Type Constructor Translation and Proof Types

The translation of type constructors is defined mutually inductively with the definition of proof kinds. The translation of the constructor c is denoted $|c|$, and $\varphi(c, \kappa)$ denotes the type of proofs that c (a target constructor) has source kind κ (more precisely, that c is the translation of a source constructor of kind κ). As we saw in the examples above, polymorphic functions are translated by adding an additional proof argument and *Top* is translated to *unit*. The translation of type constructors and the definition of proof types appears in Figure 4, where $a[b/z]$ denotes the capture-avoiding substitution of b for z in a.

A context is translated by mapping the kind and constructor translations over its bindings and adding additional bindings to hold kinding proofs, as shown in Figure 5 (where the translation of Γ is denoted $\overline{\Gamma}$). For convenience we assume that a subset of the term variables have been set aside as *proof variables* and are designated $\hat{\alpha}$ for each constructor variable α.

Proposition 1 *1. If* $\Gamma \vdash_{\mathcal{S}}$ context *then* $\overline{\Gamma} \vdash_{\mathcal{T}}$ context.

2. If $\Gamma \vdash_{\mathcal{S}} c : \kappa$ *then* $\overline{\Gamma} \vdash_{\mathcal{T}} |c| : \overline{\kappa}$.

With these translations in hand, we can now give the type-directed translation for terms. The system consists of three judgements plus the $\Gamma \vdash_{\mathcal{S}}$ context judgement of the

$$\begin{array}{rcl}\overline{\bullet} & = & \bullet\\ \overline{\Gamma[\alpha : \kappa]} & = & \overline{\Gamma}[\alpha : \overline{\kappa}][\hat{\alpha} : \varphi(\alpha,\kappa)]\\ \overline{\Gamma[x : \tau]} & = & \overline{\Gamma}[x : |\tau|]\end{array}$$

Figure 5: Context Translation

source. For each rule in the source, there is a corresponding translation rule, so the translation procedure is straightforward.

If $\Gamma \vdash_{\mathcal{S}} e : \tau$, then the *term translation* judgement

$$\Gamma \vdash e : \tau \Rightarrow e_t$$

computes e_t, the translation of e. To implement the subsumption rule, the term translation uses another translation judgement to compute kinding proofs: If $\Gamma \vdash_{\mathcal{S}} c : \kappa$, the *proof synthesis* judgement

$$\Gamma \vdash c : \kappa \Rightarrow p$$

computes p, a proof that c has kind κ. When $\Gamma \vdash c_1 \leq c_2$, the kinding proof computed is a coercion $c_1 \rightarrow c_2$ that can be used by the subsumption rule. In addition to these two judgements, a third judgement is required to handle subkinding subsumption: If $\Gamma \vdash_{\mathcal{S}} \kappa_1 \preceq \kappa_2$, the *proof coercion* judgement

$$\Gamma \vdash \kappa_1 \preceq \kappa_2 \Rightarrow pc$$

computes pc, a proof coercion from proofs of membership in κ_1 to proofs of membership in κ_2. That is, if p is a proof that c has kind κ_1, then $pc\,[c]\,p$ is a proof that c has kind κ_2.

Selected translation rules appear in Figure 6; the full system is given in Appendix A. Following are two propositions expressing static properties of the translation. Proposition 2 says that translations always exist for well-formed terms (and kinding and subkinding judgements). Proposition 3 (taken with Proposition 1) says that the translation is type correct (in well-formed types and contexts), so it satisfies our implementational demands. Although it seems clear that this translation is "right," at this point we can only trust our intuition. In the next section we address this issue a bit more carefully, and in Section 5 we give one formal sense in which the translation is right: the source and target calculi can be given a common semantics that is preserved by the translation.

Proposition 2 *1. $\Gamma \vdash_{\mathcal{S}} e : \tau$ iff $\Gamma \vdash e : \tau \Rightarrow e_t$ for some e_t.*

2. $\Gamma \vdash_{\mathcal{S}} c : \kappa$ iff $\Gamma \vdash c : \kappa \Rightarrow p$ for some p.

3. $\Gamma \vdash_{\mathcal{S}} \kappa_1 \preceq \kappa_2$ iff $\Gamma \vdash \kappa_1 \preceq \kappa_2 \Rightarrow pc$ for some pc.

Proposition 3 *1. If $\Gamma \vdash e : \tau \Rightarrow e_t$ then $\overline{\Gamma} \vdash_{\mathcal{T}} e_t : |\tau|$.*

2. If $\Gamma \vdash c : \kappa \Rightarrow p$ then $\overline{\Gamma} \vdash_{\mathcal{T}} p : \varphi(|c|, \kappa)$.

3. If $\Gamma \vdash \kappa_1 \preceq \kappa_2 \Rightarrow pc$ and $\Gamma \vdash_{\mathcal{S}} c : \kappa_1$ and $\overline{\Gamma} \vdash_{\mathcal{T}} p : \varphi(|c|, \kappa_1)$ then $\overline{\Gamma} \vdash_{\mathcal{T}} pc\,[|c|]\,p : \varphi(|c|, \kappa_2)$.

4 Type-Theoretic Implementation of Subsumption

In Section 3 we gave a translation that eliminated subsumption. This translation suffices for our implementational needs, but it is possible that another such translation could exist. Is there any reason to prefer this one over other possible translations? In this section we answer this question in the affirmative by giving a type-theoretic construction showing that our translation is the natural one.

The construction begins by building the kinds of our source calculus in the type theory of Nuprl [13] (although any sufficiently rich type theory will suffice) using the inclusionary interpretation of subtyping that $\tau_1 \leq \tau_2$ exactly when all members of τ_1 are members of τ_2. We then reinterpret subtyping using coercion-existence and make the necessary changes to our construction to maintain constructivity. This second construction, when expressed in terms of our target calculus, is isomorphic to the results of the translation. From this we conclude that the translation corresponds to the minimal consequence of moving from inclusionary subtyping (the programmer's model) to coercion-existence subtyping (the machine's model), and it is in this sense that the translation is the natural one.

In the inclusionary interpretation, $\tau_1 \leq \tau_2$ when every member of τ_1 is a member of τ_2. Another way of saying this is that $\tau_1 \leq \tau_2$ exactly when the identity function $(\lambda x.x)$ is a function from τ_1 to τ_2. With this in mind, we can define the power kind using a set type:[1]

$$[\![\mathcal{P}(\tau)]\!] = \{\alpha : \mathit{Type} \mid \alpha \leq \tau\}$$

The set type $\{z : S \mid P[z]\}$ contains all elements z of type S such that $P[z]$ is true (*i.e.,* inhabited), but suppresses the computational content in the proofs of P. Thus if $e \in \{z : S \mid P[z]\}$, then $P[e]$ is true (inhabited), but the inhabitant of $P[e]$ is inaccessible and cannot be used for computation. Additional details are available in Constable [14] and Constable *et al.* [13].

Under our current interpretation of subtyping ($\tau_1 \leq \tau_2$ iff $\lambda x.x \in \tau_1 \rightarrow \tau_2$), the computational content of a subtyping proposition is always the identity function, which, although suppressed by the set type in the power kind definition, can trivially be reconstructed. Thus there is no problem implementing subsumption constructively: Suppose we are given that τ_1 has kind $\mathcal{P}(\tau_2)$ and e has type τ_1; then we can conclude that $\lambda x.x \in \tau_1 \rightarrow \tau_2$ and hence that $e = (\lambda x.x)e \in \tau_2$.

Now suppose we wish to weaken our interpretation of subtyping to allow $\tau_1 \leq_{\text{new}} \tau_2$ whenever there is any "semantics-preserving"[2] function from τ_1 to τ_2, not necessarily the identity function. (We use $\tau_1 \dot{\rightarrow} \tau_2$ to denote the type of semantics-preserving functions.) We might naively attempt to define the power kind in the same manner as before:

$$[\![\mathcal{P}(\tau)]\!]_{\text{new}} = \{\alpha : \mathit{Type} \mid \alpha \leq_{\text{new}} \tau\} = \{\alpha : \mathit{Type} \mid \alpha \dot{\rightarrow} \tau\} \quad \text{(wrong)}$$

Unfortunately, the computational content of a subtyping proposition is now non-trivial. When τ_1 has kind $\mathcal{P}(\tau_2)$, there is no way of reconstructing the inaccessible witness of $\tau_1 \dot{\rightarrow} \tau_2$ (even though we know some such witness exists) and hence subsumption cannot be implemented constructively. This is identical to the basic problem with implementing bounded quantification discussed in the introduction. To make subsumption constructive again, we must use

[1] In Nuprl's type theory, *Type* is not a member of *Type*, but belongs instead to a higher universe, so this definition of power kinds will not strictly speaking be adequate to give a full account of our source calculus's impredicative types. The techniques of Mendler [27] could extend this construction to impredicative types, but the extension is omitted here for the sake of simplicity, since this detail does not affect our purposes in any significant way.

[2] We leave open the definition of semantics-preserving functions, although any proper definition would include the identity functions. This gives greater flexibility to the construction. The reader bothered by this may take all strict total functions to be semantics preserving. Other reasonable choices include homomorphisms, injective homomorphisms, and linear-time functions.

$$(\forall\ \mathrm{I})\frac{\Gamma[\alpha:\kappa]\vdash e:c\Rightarrow e_t}{\Gamma\vdash\Lambda\alpha{:}\kappa.e:\forall\alpha{:}\kappa.c\Rightarrow\Lambda\alpha{:}\overline{\kappa}.\,\lambda\hat{\alpha}{:}\varphi(\alpha,\kappa).\,e_t}\qquad(\forall\ \mathrm{E})\frac{\Gamma\vdash e:\forall\alpha{:}\kappa.c_1\Rightarrow e_t\qquad\Gamma\vdash c_2:\kappa\Rightarrow p}{\Gamma\vdash e\,[c_2]:c_1[c_2/\alpha]\Rightarrow e_t\,[|c_2|]\,p}$$

$$(\textsc{Subsume})\frac{\Gamma\vdash e:c_1\Rightarrow e_t\qquad\Gamma\vdash c_1:\mathcal{P}(c_2)\Rightarrow p}{\Gamma\vdash e:c_2\Rightarrow p\,e_t}\qquad(\textsc{Var})\frac{\Gamma\vdash_{\mathcal{S}}\text{context}}{\Gamma\vdash\alpha:\kappa\Rightarrow\hat{\alpha}}\quad[\alpha:\kappa]\in\Gamma$$

$$(\{\}\leq)\frac{\Gamma\vdash c_i:\mathcal{P}(c_i')\Rightarrow p_i\quad(\text{for }1\leq i\leq n)\qquad\Gamma\vdash_{\mathcal{S}}c_i:\mathit{Type}\quad(\text{for }n+1\leq i\leq m)}{\Gamma\vdash\{\ell_i:c_i{}^{[i=1\ldots m]}\}:\mathcal{P}(\{\ell_i:c_i'{}^{[i=1\ldots n]}\})\Rightarrow\lambda x{:}\{\ell_i:|c_i|^{[i=1\ldots m]}\}.\,\{\ell_i=p_i\,\pi_{\ell_i}(x)^{[i=1\ldots n]}\}}\quad n\leq m,m>0$$

$$(\to\leq)\frac{\Gamma\vdash c_1':\mathcal{P}(c_1)\Rightarrow p_1\qquad\Gamma\vdash c_2:\mathcal{P}(c_2')\Rightarrow p_2}{\Gamma\vdash c_1\to c_2:\mathcal{P}(c_1'\to c_2')\Rightarrow\lambda f{:}|c_1|\to|c_2|.\,p_2\circ f\circ p_1}\qquad(\mathit{void}\leq)\frac{\Gamma\vdash_{\mathcal{S}}c:\mathit{Type}}{\Gamma\vdash\mathit{void}:\mathcal{P}(c)\Rightarrow\lambda x{:}\mathit{void}.\,\mathit{any}_c(x)}$$

$$(\Pi\ \mathrm{I})\frac{\Gamma[\alpha:\kappa_1]\vdash c:\kappa_2\Rightarrow p}{\Gamma\vdash\lambda\alpha{:}\kappa_1.c:\Pi\alpha{:}\kappa_1.\kappa_2\Rightarrow\Lambda\alpha{:}\overline{\kappa_1}.\,\lambda\hat{\alpha}{:}\varphi(\alpha,\kappa_1).\,p}\qquad(\Pi\ \mathrm{E})\frac{\Gamma\vdash c_1:\Pi\alpha{:}\kappa_1.\kappa_2\Rightarrow p_1\qquad\Gamma\vdash c_2:\kappa_1\Rightarrow p_2}{\Gamma\vdash c_1[c_2]:\kappa_2[c_2/\alpha]\Rightarrow p_1\,[|c_2|]\,p_2}$$

$$(\Sigma\ \mathrm{I})\frac{\Gamma\vdash c_1:\kappa_1\Rightarrow p_1\qquad\Gamma\vdash c_2:\kappa_2[c_1/\alpha]\Rightarrow p_2\qquad\Gamma[\alpha:\kappa_1]\vdash_{\mathcal{S}}\kappa_2\ \text{kind}}{\Gamma\vdash\langle c_1,c_2\rangle:\Sigma\alpha{:}\kappa_1.\kappa_2\Rightarrow\langle p_1,p_2\rangle}$$

$$(\textsc{Subkind})\frac{\Gamma\vdash c:\kappa_1\Rightarrow p\qquad\Gamma\vdash\kappa_1\preceq\kappa_2\Rightarrow pc}{\Gamma\vdash c:\kappa_2\Rightarrow pc\,[|c|]\,p}\qquad(\mathcal{P}\preceq)\frac{\Gamma\vdash c_1:\mathcal{P}(c_2)\Rightarrow p}{\Gamma\vdash\mathcal{P}(c_1)\preceq\mathcal{P}(c_2)\Rightarrow\Lambda\alpha{:}\mathit{Type}.\,\lambda p'{:}\alpha\to c_1.\,p\circ p'}$$

Figure 6: Selected Translation Rules

a dependent sum instead, which does not suppress computational content:

$$[\![\mathcal{P}(\tau)]\!]_{\mathrm{new}}=\Sigma\alpha{:}\mathit{Type}.\,\alpha\dot{\to}\tau$$

The inhabitants of this sum are pairs containing the type itself and the semantics-preserving function. With this definition of the power kind, subsumption can again be implemented constructively.

Now consider the polymorphic function $\Lambda\alpha{:}\mathcal{P}(\tau).\,e$. Under the new construction, this is interpreted

$$\begin{array}{rcl}\lambda\alpha'{:}[\![\mathcal{P}(\tau)]\!].\,e' & = & \lambda\alpha'{:}(\Sigma\alpha{:}\mathit{Type}.\,\alpha\dot{\to}\tau).\,e'\\ & \cong & \lambda\alpha{:}\mathit{Type}.\,\lambda\hat{\alpha}{:}\alpha\dot{\to}\tau.\,e'[\langle\alpha,\hat{\alpha}\rangle/\alpha']\end{array}$$

where e' is the interpretation of e. This, when stated in terms of our target calculus (where all coercions are taken to be semantics preserving), is precisely the result of the translation.

If the construction is extended to higher kinds in the obvious manner and the reasonable restriction is made that type constructors may not depend upon kinding proofs (which cannot happen in our source or target calculi anyway), then this correspondence holds at higher kinds as well. That is, for all valid kinds κ, $[\![\kappa]\!]$ is isomorphic to $\Sigma\alpha{:}\overline{\kappa}.\,\varphi(\alpha,\kappa)$.[3] Thus the polymorphic function $\Lambda\alpha{:}\kappa.e$ is interpreted

$$\begin{array}{rcl}\lambda\alpha'{:}[\![\kappa]\!].e' & \cong & \lambda\alpha'{:}(\Sigma\alpha{:}\overline{\kappa}.\,\varphi(\alpha,\kappa)).\,e'\\ & \cong & \lambda\alpha{:}\overline{\kappa}.\,\lambda\hat{\alpha}{:}\varphi(\alpha,\kappa).\,e'[\langle\alpha,\hat{\alpha}\rangle/\alpha']\end{array}$$

which again is precisely the result of the translation.

5 Reconciling Inclusion and Coercion-Existence

To this point it may appear that the two interpretations of subtyping, inclusion and coercion-existence, are disparate ones. In this section we show that this need not be so, by giving a common semantics for the two interpretations. Since there is insufficient space in this paper to give a mathematically rigorous presentation of this semantics, here we simply outline the most salient points; a formal treatment appears in Crary [17].

We begin by supposing that we are given a value-respecting partial equivalence relation (VPER, defined below) semantics built over some evaluation function [21, 1, 2], and that we are given a set of coercions operating on that semantics' primitive types. We will then build a new VPER semantics that incorporates those coercions.

A partial equivalence relation (PER) is a symmetric and transitive relation. If $\simeq$ is a PER, its domain, denoted $\mathrm{Dom}(\simeq)$, is defined as the set of terms e such that $e\simeq e$. Thus a PER is an equivalence relation over its domain. A PER is value-respecting if evaluation respects equivalence classes, that is

- $e\simeq e$ and $e\Downarrow v$ implies $e\simeq v$
- $e\simeq e'$ and $e\Downarrow v$ implies $e'\Downarrow v'$

where we write $e\Downarrow v$ if e evaluates to v.[4] A VPER semantics associates each type τ with a VPER $\simeq_\tau$. The members of τ are those terms in $\mathrm{Dom}(\simeq_\tau)$, and equality on τ is given by $\simeq_\tau$.

The subtyping relationship stems from two sources: subtyping on primitive types and type constructors (*e.g.,* $\mathit{int}\leq\mathit{real}$ and $\mathit{list}[\tau]\leq\mathit{tree}[\tau]$), and the induced subtyping relations over the type constructors. Coercions resulting from the latter do nothing of computational interest except apply primitive coercions resulting from the former [5], so we will give the construction of the new semantics for primitive types and allow its extension to type constructors to be interleaved with it.

We construct the new semantics by induction on the subtyping order resulting from our set of coercions. Let σ and

[3] In fact, an examination of the type $\Sigma\alpha{:}\overline{\kappa}.\,\varphi(\alpha,\kappa)$ and the type $[\![\kappa]\!]$ reveals that the former is essentially a phase-split [22] version of the latter.

[4] The second clause dictates that a convergent expression shall not be equal to a divergent or "stuck" expression. The conventional definition of value-respecting omits this requirement.

τ be primitive types and let f be a coercion $\sigma \rightarrow \tau$. Let $\simeq_\tau$ be the VPER for τ under the old semantics and (invoking induction) let $\simeq'_\sigma$ be the VPER for σ under the new semantics. We assume that $\mathrm{Dom}(\simeq_\tau)$ and $\mathrm{Dom}(\simeq'_\sigma)$ are disjoint; if necessary, this can be achieved by tagging each primitive type before we begin. The new VPER for τ, $\simeq'_\tau$, is the union of $\simeq_\tau$ and $\simeq'_\sigma$ that condenses equivalence classes to respect the coercion f. Formally, $\simeq'_\tau$ is the least symmetric, transitive relation such that:

- $e \simeq_\tau e'$ implies $e \simeq'_\tau e'$
- $e \simeq'_\sigma e'$ implies $e \simeq'_\tau e'$
- $e \simeq'_\sigma e$ implies $e \simeq'_\tau f\, e$

Note that $\mathrm{Dom}(\simeq'_\tau) = \mathrm{Dom}(\simeq_\tau) \uplus \mathrm{Dom}(\simeq'_\sigma)$. If there are additional types σ' with coercions $\sigma' \rightarrow \tau$ then this construction is iterated over those other coercions as well.

To complete the new VPER semantics we must update the evaluation function to incorporate coercions in order for the VPERs of our semantics to indeed be value respecting; otherwise the evaluation of one term could become "stuck" while that of an equivalent term did not. For instance, following the example in Section 1, suppose f is a coercion $int \rightarrow real$ and suppose the original evaluation system contains the rule:

$$\frac{e_1 \Downarrow \mathtt{sqrt} \qquad e_2 \Downarrow r}{e_1 e_2 \Downarrow \sqrt{r}} \quad r \in \mathbb{R}$$

Then the new evaluation system must add the rule:

$$\frac{e_1 \Downarrow \mathtt{sqrt} \qquad e_2 \Downarrow z \qquad f\, z \Downarrow r}{e_1 e_2 \Downarrow \sqrt{r}} \quad z \in \mathbb{Z}, r \in \mathbb{R}$$

It remains to make a few observations about the new semantics: First, in the new semantics, coercions *are* identity functions, which reconciles the two interpretations of subtyping. Second, our augmentation of the evaluation system amounts to adding "run-time" coercions. In that semantic context, the translation in this paper transforms a program that might need run-time coercions to one that will not. Third, the new semantics does not collapse any equivalence classes that exist in the original semantics. This is easily shown from the fact that our coercion set forms a forest. If our coercion set were permitted to form a DAG then it would be necessary to require that our primitive coercions be coherent among themselves [29].

5.1 Coherence

Since the translation of a term depends upon the derivation of its typing judgement, it is possible for a term to have many different possible translations. If we wish our programming language to be portable (and do not wish to fix a particular typing strategy in the language definition), it is important that any two such translations be semantically equivalent. In the first-order case, without polymorphic functions, the above semantics gives us this result: Coercions are built entirely of primitive coercions, and primitive coercions are identity functions in the semantics (or the extension of identity functions to higher kinds). Thus any coercion used in subsumption is an identity function, and hence coherence.

However, a complication appears in the presence of polymorphic functions. Consider the function $\Lambda\alpha{\leq}Top.\, \lambda x{:}\alpha.\, (x : Top)$. This has two different translations, depending upon whether the Var or TopSub rule is used to show $\alpha \leq Top$:

- $\Lambda\alpha{:}Type.\, \lambda\hat{\alpha}{:}\alpha \rightarrow unit.\, \lambda x{:}\alpha.\, \hat{\alpha}\, x$
- $\Lambda\alpha{:}Type.\, \lambda\hat{\alpha}{:}\alpha \rightarrow unit.\, \lambda x{:}\alpha.\, (\lambda x{:}\alpha.\diamond)x$

These actually are equivalent, but that is only an artifact of the restricted type system in which we are presently working. In a language with nontermination or computational effects, $\hat{\alpha}$ could be instantiated with any sort of pathological function, which would make the two terms inequivalent. This is the direct consequence of allowing an arbitrary argument for the kinding proof.[5] (In the terminology of Section 4, the problem is that such functions are not semantics preserving.)

In practice, of course, kinding proofs are instantiated only by the compiler and never with this sort of pathological argument. To get a formal coherence result, we must restrict proof argument types so that they include only the sort of arguments they will actually receive. With such a restriction in place, all proof arguments will be identity functions, like the primitive coercions. Then, as in the first-order case, coercions used in subsumption will be identity functions, and again coherence results. The necessary restriction can easily be implemented syntactically by adding a type of coercions which can only be built with specified combinators, as in the BCGS system of variant types.

6 Related Work

In the seminal paper on subtyping as coercion-existence [4], BCGS developed the use of coercions to eliminate subsumption in the second-order case. Their focus was on making a wider array of semantic techniques available to model subtyping calculi. To that end, they proved a syntactic coherence result, leaving the semantics of the target calculus open (except to the extent to which it was constrained by the equational theory). The translation presented in this paper is the generalization of their translation to higher-order subtyping; in particular, the coercions used in BCGS are a special case of the kinding proofs used in this paper. Curien and Ghelli [18] developed another technique for showing the syntactic coherence of a similar second-order translation by proof rewriting. Breazu-Tannen, Gunter and Scedrov [5] gave an operational semantics to a first-order special case (without polymorphism, bounded quantification or subtyping on primitive types) and showed that the translation in that case did not observably affect the computation.

Bruce and Longo [6] developed a model of subtyping and bounded quantification that interprets types as PERs, but interprets an element of type τ as an equivalence class of $\simeq_\tau$, not as a member of $\mathrm{Dom}(\simeq_\tau)$. Under this interpretation, equality in a type corresponds to equality in the model, in contrast to our semantics where equality in τ corresponds to equivalence under $\simeq_\tau$; moreover, in the Bruce and Longo model the interpretation of a term depends upon the type

[5] BCGS, for their system without variant types, show coherence of a translation that permits arbitrary arguments, but they are able to do this because they include the rule

$$\frac{e : unit}{e = \diamond}$$

which is questionable in the presence of divergent expressions, and certainly unsound in the presence of effects.

it is placed in. Thus, a member of a subtype is not (in the model) a member of the supertype, so coercions are required to interpret subtyping; this is handled in the model by a coherent translation. In some sense, then, the model of Bruce and Longo is the exact opposite of the semantics in Section 5: In this paper we begin with a set of coercions and build a PER semantics that permits subtyping by inclusion; Bruce and Longo begin with a PER and subtyping by inclusion (and refinement) and with it build a model that uses coercions.

Also closely related to this work is the work of Jones on qualified types [24, 23], which are types required to obey some predicate. These predicates are used to implement type classes in the Gofer programming language [26]. As here, Jones implements qualified types using a translation motivated by the propositions-as-types principle. This translation passes evidence terms analogous to the kinding proofs of this paper. Jones shows also that the translation is coherent whenever terms have unique minimal types [23, 25].

Early work in this area was done by Reynolds [29], who used category theory to develop a framework for defining semantics for and showing the coherence of calculi with subtyping and overloaded operators. More recently, the semantics for the programming language Forsythe [31] interpreted intersection types as a category-theoretic pullback, a non-syntax-directed definition; this was proven coherent in Reynolds [30].

7 Summary and Conclusions

We have given a translation of a higher-order subtyping calculus into subsumption-free calculus suitable for implementation by conventional means. Such a translation is necessitated by changing the interpretation of subtyping from inclusion (the programmer's model) to coercion-existence (the machine's model). A type-theoretic construction shows that our translation is the minimal consequence of that shift in interpretation. Finally, we have reconciled the two interpretations in a common semantics. Coherence relative to that semantics is easily shown. That ease is a consequence of the fact, revealed by the type-theoretic construction, that the translation does not actually change very much.

The type-theoretic construction also gives a framework for working out generalizations to more expressive type systems. In particular, the translation easily generalizes to the dependent record kinds of λ^K [16]. That generalized translation is put to practical use in the KML compiler [15]. We also claim, without elaboration, that the construction shows how to generalize the translation to Cardelli's formulation of $F^\omega_\leq$ with monotone kinds [8].

We also hope that the reconciliation of Section 5 can be used to manage complexity in formalized mathematics. In informal mathematics, a structure in algebra or analysis is often used as a substructure of another, implicitly using isomorphism. These isomorphisms must be included explicitly in formalized mathematics, and lead to considerable clutter. We hope that a construction as in Section 5 can be used to hide such coercions of isomorphism.

References

[1] Stuart Allen. A non-type-theoretic definition of Martin-Löf's types. In *Second IEEE Symposium of Logic in Computer Science*, pages 215–221, Ithaca, New York, June 1987.

[2] Stuart Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1987.

[3] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[4] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[5] Val Breazu-Tannen, Carl A. Gunter, and Andre Scedrov. Computing with coercions. In *1990 ACM Conference on Lisp and Functional Programming*, pages 44–60, 1990.

[6] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1991.

[7] Luca Cardelli. Structural subtyping and the notion of power type. In *Fifteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 70–79, San Diego, January 1988.

[8] Luca Cardelli. Notes about $\mathrm{F}^\omega_{<:}$. Unpublished Manuscript, 1990.

[9] Luca Cardelli. Typeful programming. In *Formal Description of Programming Concepts.* Springer-Verlag, 1991.

[10] Luca Cardelli. An implementation of $F_{<:}$. SRC Research Report 97, Digital Equipment Corporation, Systems Research Center, February 1993.

[11] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In *Theoretical Aspects of Computer Software 1991*, volume 526 of *Lecture Notes in Computer Science*, pages 750–770, Sendai, Japan, 1991. Springer-Verlag.

[12] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[13] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[14] Robert L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. In *Topics in the Theory of Computation*, volume 24 of *Annals of Discrete Mathematics*, pages 21–37. Elsevier, 1985. Selected papers of the International Conference on Foundations of Computation Theory 1983.

[15] Karl Crary. *KML Reference Manual.* Department of Computer Science, Cornell University, 1996.

[16] Karl Crary. A unified framework for modules and objects and its application to programming language design. Technical report, Department of Computer Science, Cornell University, 1996.

[17] Karl Crary. Semantic reconciliation of subtyping by inclusion and coercion. Technical report, Department of Computer Science, Cornell University, 1997. Forthcoming.

[18] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. In *Fifteenth Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages 132–146. Springer-Verlag, 1990.

[19] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[20] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[21] Robert Harper. Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, 14:71–84, 1992.

[22] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[23] Mark P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Oxford University Computing Laboratory, July 1992.

[24] Mark P. Jones. A theory of qualified types. In *Fourth European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, Rennes, France, 1992. Springer-Verlag.

[25] Mark P. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, Department of Computer Science, September 1993.

[26] Mark P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, May 1994.

[27] Paul Francis Mendler. *Inductive Definition in Type Theory.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, September 1987.

[28] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 1997. To appear. Available as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94.

[29] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, 1980.

[30] John C. Reynolds. The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software 1991*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Sendai, Japan, 1991. Springer-Verlag.

[31] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-94-146, Carnegie Mellon University, School of Computer Science, June 1996.

A Source and Translation Rules

Since there is a one-to-one correspondence between rules in the source calculus and translation rules, we present the two simultaneously in the interest of brevity. What follow are the translation rules. For any translation rule, the corresponding rule in the source calculus is obtained by dropping the extract "$\Rightarrow e$" from each judgement with an extract in the rule. The context formation judgements have no extract and thus are considered as judgements in the source calculus without modification. Moreover, some translation rules ignore the extracts of some antecedent judgements (usually because those extracts are trivial); such judgements are stated in source calculus form, without the ignored extracts. Also, the definition of $\beta\eta$-equivalence is completely standard and is omitted (also in Appendix B).

A.1 Judgement Forms

$c_1 =_{\beta\eta} c_2$	c_1 and c_2 are $\beta\eta$-equivalent
$\Gamma \vdash \text{context}$	Γ is a valid context
$\Gamma \vdash \kappa_1 \preceq \kappa_2 \Rightarrow pc$	κ_1, κ_2 are valid kinds and κ_1 is a subkind of κ_2 (with proof coercion pc)
$\Gamma \vdash c : \kappa \Rightarrow p$	c had kind κ (with proof p)
$\Gamma \vdash e : c \Rightarrow e_t$	e has type c (with translation e_t)

A.2 Abbreviations

$$\begin{aligned}
\mathit{Type} &\stackrel{\text{def}}{=} \mathcal{P}(\mathit{Top}) \\
\Gamma \vdash \kappa \text{ kind} &\stackrel{\text{def}}{=} \Gamma \vdash \kappa \preceq \kappa \\
\Gamma \vdash c_1 \leq c_2 \Rightarrow p &\stackrel{\text{def}}{=} \Gamma \vdash c_1 : \mathcal{P}(c_2) \Rightarrow p \\
\text{let } \alpha = c \text{ and } x : c' = e \text{ in } e' &\stackrel{\text{def}}{=} (\lambda x{:}c'.\, e'[c/\alpha])\, e
\end{aligned}$$

A.3 Context Formation

$$\frac{}{\bullet \vdash \text{context}} \qquad (\text{NullCtx})$$

$$\frac{\Gamma \vdash \kappa \text{ kind}}{\Gamma[\alpha : \kappa] \vdash \text{context}} \quad \alpha \notin \Gamma \qquad (\text{KindCtx})$$

$$\frac{\Gamma \vdash c : \mathit{Type}}{\Gamma[x : c] \vdash \text{context}} \quad x \notin \Gamma \qquad (\text{TypeCtx})$$

A.4 Kind Formation and Subkinding

$$\frac{\Gamma \vdash \kappa_1 \preceq \kappa_2 \Rightarrow pc \qquad \Gamma \vdash \kappa_2 \preceq \kappa_3 \Rightarrow pc'}{\Gamma \vdash \kappa_1 \preceq \kappa_3 \Rightarrow \Lambda\alpha{:}\overline{\kappa_1}.\, \lambda p{:}\varphi(\alpha,\kappa_1).\, pc'\,[\alpha]\,(pc\,[\alpha]\,p)} \quad (\textsc{Trans})$$

$$\frac{\begin{array}{c}\Gamma \vdash \kappa_1' \preceq \kappa_1 \Rightarrow pc_1 \\ \Gamma[\alpha:\kappa_1'] \vdash \kappa_2 \preceq \kappa_2' \Rightarrow pc_2 \qquad \Gamma[\alpha:\kappa_1] \vdash \kappa_2 \text{ kind}\end{array}}{\begin{array}{l}\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \preceq \Pi\alpha{:}\kappa_1'.\kappa_2' \Rightarrow \\ \quad \Lambda\beta{:}\overline{\kappa_1} \to \overline{\kappa_2}.\, \lambda p{:}\varphi(\beta, \Pi\alpha{:}\kappa_1.\kappa_2). \\ \qquad \Lambda\alpha{:}\overline{\kappa_1'}.\, \lambda\hat{\alpha}{:}\varphi(\alpha,\kappa_1').\, pc_2\,[\beta\alpha]\,(p\,[\alpha]\,(pc_1\,[\alpha]\,\hat{\alpha}))\end{array}} \quad (\textsc{Pi})$$

$$\frac{\begin{array}{c}\Gamma \vdash \kappa_1 \preceq \kappa_1' \Rightarrow pc_1 \\ \Gamma[\alpha:\kappa_1] \vdash \kappa_2 \preceq \kappa_2' \Rightarrow pc_2 \qquad \Gamma[\alpha:\kappa_1'] \vdash \kappa_2' \text{ kind}\end{array}}{\begin{array}{l}\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \preceq \Sigma\alpha{:}\kappa_1'.\kappa_2' \Rightarrow \\ \quad \Lambda\beta{:}\overline{\kappa_1} \times \overline{\kappa_2}.\, \lambda p{:}\varphi(\beta, \Sigma\alpha{:}\kappa_1.\kappa_2). \\ \qquad \langle pc_1\,[\pi_1(\beta)]\,\pi_1(p), \\ \qquad\; \text{let } \alpha = \pi_1(\beta) \text{ and } \hat{\alpha} : \varphi(\pi_1(\beta),\kappa_1) = \pi_1(p) \text{ in} \\ \qquad\quad pc_2\,[\pi_2(\beta)]\,\pi_2(p)\rangle\end{array}} \quad (\textsc{Sigma})$$

$$\frac{\Gamma \vdash c_1 \leq c_2 \Rightarrow p}{\Gamma \vdash \mathcal{P}(c_1) \preceq \mathcal{P}(c_2) \Rightarrow \Lambda\alpha{:}\mathit{Type}.\, \lambda p'{:}\alpha \to c_1.\, \lambda x{:}\alpha.\, p\,(p'\,x)} \quad (\textsc{Pow})$$

A.5 Kinding

$$\frac{\Gamma \vdash \text{context}}{\Gamma \vdash \alpha : \kappa \Rightarrow \hat{\alpha}} \quad \alpha : \kappa \in \Gamma \quad (\textsc{TypeVar})$$

$$\frac{\Gamma[\alpha:\kappa_1] \vdash c : \kappa_2 \Rightarrow p}{\Gamma \vdash \lambda\alpha{:}\kappa_1.c : \Pi\alpha{:}\kappa_1.\kappa_2 \Rightarrow \Lambda\alpha{:}\overline{\kappa_1}.\, \lambda\hat{\alpha}{:}\varphi(\alpha,\kappa_1).p} \quad (\textsc{PiIntro})$$

$$\frac{\Gamma \vdash c_1 : \Pi\alpha{:}\kappa_1.\kappa_2 \Rightarrow p_1 \qquad \Gamma \vdash c_2 : \kappa_1 \Rightarrow p_2}{\Gamma \vdash c_1[c_2] : \kappa_2[c_2/\alpha] \Rightarrow p_1\,[|c_2|]\,p_2} \quad (\textsc{PiElim})$$

$$\frac{\Gamma[\alpha:\kappa_1] \vdash c[\alpha] : \kappa_2 \Rightarrow p}{\Gamma \vdash c : \Pi\alpha{:}\kappa_1.\kappa_2 \Rightarrow \Lambda\alpha{:}\overline{\kappa_1}.\, \lambda\hat{\alpha}{:}\varphi(\alpha,\kappa_1).p} \quad \alpha \notin c \quad (\textsc{PiEta})$$

$$\frac{\begin{array}{c}\Gamma \vdash c_1 : \kappa_1 \Rightarrow p_1 \\ \Gamma \vdash c_2 : \kappa_2[c_1/\alpha] \Rightarrow p_2 \qquad \Gamma[\alpha:\kappa_1] \vdash \kappa_2 \text{ kind}\end{array}}{\Gamma \vdash \langle c_1, c_2\rangle : \Sigma\alpha{:}\kappa_1.\kappa_2 \Rightarrow \langle p_1, p_2\rangle} \quad (\textsc{SigmaIntro})$$

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2 \Rightarrow p}{\Gamma \vdash \pi_1(c) : \kappa_1 \Rightarrow \pi_1(p)} \quad (\textsc{SigmaElim1})$$

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2 \Rightarrow p}{\Gamma \vdash \pi_2(c) : \kappa_2[\pi_1(c)/\alpha] \Rightarrow \pi_2(p)} \quad (\textsc{SigmaElim2})$$

$$\frac{\begin{array}{c}\Gamma \vdash \pi_1(c) : \kappa_1 \Rightarrow p_1 \\ \Gamma \vdash \pi_2(c) : \kappa_2[\pi_1(c)/\alpha] \Rightarrow p_2 \qquad \Gamma[\alpha:\kappa_1] \vdash \kappa_2 \text{ kind}\end{array}}{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2 \Rightarrow \langle p_1, p_2\rangle} \quad (\textsc{SigmaEta})$$

$$\frac{\Gamma \vdash c_1 : \mathit{Type} \qquad \Gamma \vdash c_2 : \mathit{Type}}{\Gamma \vdash c_1 \to c_2 : \mathit{Type} \Rightarrow \lambda x{:}|c_1 \to c_2|.\diamond} \quad (\textsc{Arrow})$$

$$\frac{\Gamma[\alpha:\kappa] \vdash c : \mathit{Type}}{\Gamma \vdash \forall\alpha{:}\kappa.c : \mathit{Type} \Rightarrow \lambda x{:}|\forall\alpha{:}\kappa.c|.\diamond} \quad (\textsc{Quant})$$

$$\frac{\Gamma \vdash \text{context} \qquad \Gamma \vdash c_i : \mathit{Type} \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash \{\ell_i : {c_i}^{[i=1\ldots n]}\} : \mathit{Type} \Rightarrow \lambda x{:}|\{\ell_i : {c_i}^{[i=1\ldots n]}\}|.\diamond} \quad (\textsc{Record})$$

$$\frac{\Gamma \vdash \text{context}}{\Gamma \vdash \mathit{Top} : \mathit{Type} \Rightarrow \lambda x{:}\mathit{unit}.\diamond} \quad (\textsc{Top})$$

$$\frac{\Gamma \vdash \text{context}}{\Gamma \vdash \mathit{void} : \mathit{Type} \Rightarrow \lambda x{:}\mathit{void}.\diamond} \quad (\textsc{Void})$$

$$\frac{\Gamma \vdash c : \kappa_1 \Rightarrow p \qquad \Gamma \vdash \kappa_1 \preceq \kappa_2 \Rightarrow pc}{\Gamma \vdash c : \kappa_2 \Rightarrow pc\,[|c|]\,p} \quad (\textsc{Subkind})$$

A.6 Subtyping

$$\frac{\Gamma \vdash c_1 : \mathit{Type} \qquad \Gamma \vdash c_2 : \mathit{Type} \qquad c_1 =_{\beta\eta} c_2}{\Gamma \vdash c_1 \leq c_2 \Rightarrow \lambda x{:}c_1.x} \quad (\textsc{Reflex})$$

$$\frac{\Gamma \vdash c_1' \leq c_1 \Rightarrow p_1 \qquad \Gamma \vdash c_2 \leq c_2' \Rightarrow p_2}{\Gamma \vdash c_1 \to c_2 \leq c_1' \to c_2' \Rightarrow \lambda f{:}|c_1| \to |c_2|.\, \lambda x{:}|c_1'|.\, p_2\,(f\,(p_1\,x))} \quad (\textsc{ArrowSub})$$

$$\frac{\Gamma \vdash \kappa_2 \preceq \kappa_1 \Rightarrow pc \qquad \Gamma[\alpha : \kappa_2] \vdash c_1 \leq c_2 \Rightarrow p}{\begin{array}{l}\Gamma \vdash \forall\alpha{:}\kappa_1.c_1 \leq \forall\alpha{:}\kappa_2.c_2 \Rightarrow \\ \quad \lambda f{:}(\forall\alpha{:}\overline{\kappa_1}.\varphi(\alpha,\kappa_1) \to |c_1|). \\ \qquad \Lambda\alpha{:}\overline{\kappa_2}.\, \lambda\hat{\alpha}{:}\varphi(\alpha,\kappa_2).\, p\,(f\,[\alpha]\,(pc\,[\alpha]\,\hat{\alpha}))\end{array}} \quad (\textsc{QuantSub})$$

$$\frac{\begin{array}{c}\Gamma \vdash c_i \leq c_i' \Rightarrow p_i \quad (\text{for } 1 \leq i \leq n) \\ \Gamma \vdash c_i : \mathit{Type} \quad (\text{for } n+1 \leq i \leq m)\end{array}}{\begin{array}{l}\Gamma \vdash \{\ell_i : {c_i}^{[i=1\ldots m]}\} \leq \{\ell_i : {c_i'}^{[i=1\ldots n]}\} \Rightarrow \\ \quad \lambda x{:}\{\ell_i : |c_i|^{[i=1\ldots m]}\}.\, \{\ell_i = p_i\, \pi_{\ell_i}(x)^{[i=1\ldots n]}\}\end{array}} \quad \begin{array}{l} n \leq m \\ m > 0 \end{array} \quad (\textsc{RecordSub})$$

$$\frac{\Gamma \vdash c : \mathcal{P}(c')}{\Gamma \vdash c \leq \mathit{Top} \Rightarrow \lambda x{:}|c|.\diamond} \quad (\textsc{TopSub})$$

Note: The TopSub rule does not enlarge the set of provable judgements in the source calculus, so it could, strictly speaking, be omitted. However, it allows smaller proofs and extracts, so it is still useful in a practical system.

$$\frac{\Gamma \vdash c : \mathit{Type}}{\Gamma \vdash \mathit{void} \leq c \Rightarrow \lambda x{:}\mathit{void}.\, \mathit{any}_c(x)} \quad (\textsc{VoidSub})$$

A.7 Typing

$$\frac{\Gamma \vdash \text{context}}{\Gamma \vdash x : c \Rightarrow x} \quad x : c \in \Gamma \quad (\textsc{Var})$$

$$\frac{\Gamma[x : c_1] \vdash e : c_2 \Rightarrow e_t}{\Gamma \vdash \lambda x{:}c_1.e : c_1 \to c_2 \Rightarrow \lambda x{:}|c_1|.e_t} \quad (\textsc{ArrowIntro})$$

$$\frac{\Gamma \vdash e_1 : c_1 \to c_2 \Rightarrow e_{t1} \qquad \Gamma \vdash e_2 : c_1 \Rightarrow e_{t2}}{\Gamma \vdash e_1 e_2 : c_2 \Rightarrow e_{t1} e_{t2}} \quad (\textsc{ArrowElim})$$

$$\frac{\Gamma[\alpha : \kappa] \vdash e : c \Rightarrow e_t}{\Gamma \vdash \Lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.c \Rightarrow \Lambda\alpha{:}\overline{\kappa}.\, \lambda\hat{\alpha}{:}\varphi(\alpha,\kappa).\, e_t} \quad (\textsc{QuantIntro})$$

$$\frac{\Gamma \vdash e : \forall\alpha{:}\kappa.c_1 \Rightarrow e_t \qquad \Gamma \vdash c_2 : \kappa \Rightarrow p}{\Gamma \vdash e\,[c_2] : c_1[c_2/\alpha] \Rightarrow e_t\,[|c_2|]\,p} \quad (\textsc{QuantElim})$$

$$\frac{\Gamma \vdash \text{context} \qquad \Gamma \vdash e_i : c_i \Rightarrow e_{ti} \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash \{\ell_i = {e_i}^{[i=1\ldots n]}\} : \{\ell_1 : {c_1}^{[i=1\ldots n}\} \Rightarrow \{\ell_1 = {e_{t1}}^{[i=1\ldots n]}\}} \quad (\textsc{RecordIntro})$$

$$\frac{\Gamma \vdash e : \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \Rightarrow e_t}{\Gamma \vdash \pi_{\ell_i}(e) : c_i \Rightarrow \pi_{\ell_i}(e_t)} \quad 1 \leq i \leq n \quad (\textsc{RecordElim})$$

$$\frac{\Gamma \vdash e : c_1 \Rightarrow e_t \qquad \Gamma \vdash c_1 \leq c_2 \Rightarrow p}{\Gamma \vdash e : c_2 \Rightarrow p\,e_t} \quad (\textsc{Subtype})$$

B Target Rules

B.1 Judgement Forms

$c_1 =_{\beta\eta} c_2$	c_1 and c_2 are $\beta\eta$-equivalent
$\Gamma \vdash$ context	Γ is a valid context
$\Gamma \vdash c : \kappa$	c had kind κ
$\Gamma \vdash e : c$	e has type c

B.2 Context Formation

$$\frac{}{\bullet \vdash \text{context}} \quad (\textsc{NullCtx})$$

$$\frac{\Gamma \vdash \text{context}}{\Gamma[\alpha : \kappa] \vdash \text{context}} \quad \alpha \notin \Gamma \quad (\textsc{KindCtx})$$

$$\frac{\Gamma \vdash c : \mathit{Type}}{\Gamma[x : c] \vdash \text{context}} \quad x \notin \Gamma \quad (\textsc{TypeCtx})$$

B.3 Kinding

$$\frac{\Gamma \vdash \text{context}}{\Gamma \vdash \alpha : \kappa} \quad \alpha : \kappa \in \Gamma \quad (\textsc{TypeVar})$$

$$\frac{\Gamma[\alpha : \kappa_1] \vdash c : \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.c : \kappa_1 \to \kappa_2} \quad (\textsc{PiIntro})$$

$$\frac{\Gamma \vdash c_1 : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash c_2 : \kappa_1}{\Gamma \vdash c_1[c_2] : \kappa_2} \quad (\textsc{PiElim})$$

$$\frac{\Gamma \vdash c_1 : \kappa_1 \qquad \Gamma \vdash c_2 : \kappa_2}{\Gamma \vdash \langle c_1, c_2 \rangle : \kappa_1 \times \kappa_2} \qquad (\textsc{SigmaIntro})$$

$$\frac{\Gamma \vdash c : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_i(c) : \kappa_i} \quad i = 1, 2 \qquad (\textsc{SigmaElim1})$$

$$\frac{\Gamma \vdash c_1 : \mathit{Type} \qquad \Gamma \vdash c_2 : \mathit{Type}}{\Gamma \vdash c_1 \rightarrow c_2 : \mathit{Type}} \qquad (\textsc{Arrow})$$

$$\frac{\Gamma[\alpha : \kappa] \vdash c : \mathit{Type}}{\Gamma \vdash \forall \alpha{:}\kappa.c : \mathit{Type}} \qquad (\textsc{Quant})$$

$$\frac{\Gamma \vdash \text{context} \qquad \Gamma \vdash c_i : \mathit{Type} \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash \{\ell_1 : c_1, \ldots, \ell_n : c_n\} : \mathit{Type}} \qquad (\textsc{Record})$$

$$\frac{\Gamma \vdash \text{context}}{\Gamma \vdash \mathit{unit} : \mathit{Type}} \qquad (\textsc{Unit})$$

$$\frac{\Gamma \vdash \text{context}}{\Gamma \vdash \mathit{void} : \mathit{Type}} \qquad (\textsc{Void})$$

B.4 Typing

$$\frac{\Gamma \vdash \text{context}}{\Gamma \vdash x : c} \quad x : c \in \Gamma \qquad (\textsc{Var})$$

$$\frac{\Gamma[x : c_1] \vdash e : c_2}{\Gamma \vdash \lambda x{:}c_1.e : c_1 \rightarrow c_2} \qquad (\textsc{ArrowIntro})$$

$$\frac{\Gamma \vdash e_1 : c_1 \rightarrow c_2 \qquad \Gamma \vdash e_2 : c_1}{\Gamma \vdash e_1 e_2 : c_2} \qquad (\textsc{ArrowElim})$$

$$\frac{\Gamma[\alpha : \kappa] \vdash e : c}{\Gamma \vdash \Lambda \alpha{:}\kappa.e : \forall \alpha{:}\kappa.c} \qquad (\textsc{QuantIntro})$$

$$\frac{\Gamma \vdash e : \forall \alpha{:}\kappa.c_1 \qquad \Gamma \vdash c_2 : \kappa}{\Gamma \vdash e\,[c_2] : c_1[c_2/\alpha]} \qquad (\textsc{QuantElim})$$

$$\frac{\Gamma \vdash \text{context} \qquad \Gamma \vdash e_i : c_i \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 : c_1, \ldots, \ell_n : c_n\}} \qquad (\textsc{RecordIntro})$$

$$\frac{\Gamma \vdash e : \{\ell_1 : c_1, \ldots, \ell_n : c_n\}}{\Gamma \vdash \pi_{\ell_i}(e) : c_i} \quad 1 \leq i \leq n \qquad (\textsc{RecordElim})$$

$$\frac{\Gamma \vdash \text{context}}{\Gamma \vdash \diamond : \mathit{unit}} \qquad (\textsc{UnitIntro})$$

$$\frac{\Gamma \vdash e : \mathit{void} \qquad \Gamma \vdash c : \mathit{Type}}{\Gamma \vdash \mathit{any}_c(e) : c} \qquad (\textsc{VoidElim})$$

$$\frac{\Gamma \vdash e : c_1 \qquad \Gamma \vdash c_2 : \mathit{Type} \qquad \Gamma \vdash c_1 =_{\beta\eta} c_2}{\Gamma \vdash e : c_2} \qquad (\textsc{Equal})$$