Typed Compilation of Inclusive Subtyping

Karl Crary

Carnegie Mellon University

Abstract

I present a type-preserving translation that eliminates subtyping and bounded quantification without introducing any run-time costs. This translation is based on Mitchell and Pierce's encoding of bounded quantification using intersection types. I show that, previous negative observations notwithstanding, the encoding is adequate given a sufficiently rich target type theory. The necessary target type theory is made easily typecheckable by including a collection of explicit coercion combinators, which are already desired for eliminating subtyping. However, no form of coercion abstraction is necessary (even to support bounded quantification), leading to a simple target language.

1 Introduction

Type-preserving compilers, those that utilize strongly typed intermediate languages, offer several compelling advantages over untyped compilers. A typed compiler can utilize type information to enable optimizations that would otherwise be prohibitively difficult or impossible. Internal type checking can be used to help debug a compiler by catching errors introduced into programs in optimization or transformation stages. Finally, if preserved through the compiler to its ultimate output (or at least to some interchange language), types can be used to certify that executables are *safe*, that is, free of certain fatal errors or malicious behavior [8].

Typed compilation is often particularly profitable for advanced programming languages, which may be challenging to implement efficiently or correctly without exploiting types. However, advanced programming languages with sophisticated type systems pose their own challenges to typed compilation: the typing constructs of a source language must either be included in the compiler's typed intermediate languages, or be "compiled away" into more primitive constructs. Where possible, it is generally preferable to reduce sophisticated typing constructs to more primitive ones, because typed intermediate languages are often fairly complicated already without the added complexity of source language features.

In this paper I consider the typed compilation of a language supporting subtyping and bounded quantification [3, 2]. Subtyping is a pervasive language feature, in that it interacts with most other language features, and therefore can substantially complicate programming languages that include it. This is particularly true for low-level typed intermediate languages. Therefore, as is often the case, it is desirable to dismantle subtyping in favor of more primitive and easy-to-type constructs.

One well-known way to do so is the seminal "Penn interpretation" of Breazu-Tannen, *et al.* [1]. The Penn interpretation eliminates instances of subsumption by inserting explicit calls to coercion functions, and handles bounded quantification by rewriting polymorphic functions to take an additional coercion argument mapping the function's type argument to its upper bound. Although Breazu-Tannen, *et al.*'s interest was in semantics, their translation can also easily be viewed as a type-preserving compilation strategy. Indeed, under one interpretation of subtyping, Breazu-Tannen *et al.*'s translation cannot be improved upon in any essential way.

In a practical setting, subtyping can be interpreted in two different ways: *inclusively*, where the members of a subtype actually belong to the supertype, and *coercively*, in which a run-time coercion may be necessary to convert members of the subtype into members of the supertype. For coercive subtyping, the costs of the Penn interpretation are unavoidable (in general), but for inclusive subtyping, the run-time application of coercions and run-time passing of coercions to polymorphic functions represent unnecessary and unacceptable costs. In many settings, the avoidability of these costs makes inclusive subtyping more attractive than coercive. This has led many language designers to eschew language features requiring run-time coercions (such as `int` $\leq$ `float` subtyping) in favor of ones enjoying a purely inclusive interpretation, such as records with prefix subtyping and (often) objects.

What we desire, then, is a type-preserving transformation that eliminates inclusive subtyping without introducing any run-time costs. We may begin with a preliminary observation about the target language of any such transformation. If subtyping is eliminated, then subsumption must be performed explicitly, but if that explicit subsumption is to be performed without run-time cost, then it cannot be performed by the ordinary dynamic constructs of the language. Thus, our target language must include a collection of combinators for building *static* coercions, in the style of Curien and Ghelli [5], for example.

With such a collection of combinators, and in the absence of bounded quantification, it is easy to construct a static coercion to replace each instance of subsumption in the source language. However, in the presence of bounded quantification one is once again left with the obvious problem of producing coercions from quantified types to their upper bounds. One natural way to solve this problem is to introduce coercion variables and a way to abstract over them (statically, so as not to incur run-time cost), and then to abstract a new coercion variable at each polymorphic function just as in the Penn interpretation. Such an approach might be viewed as an inclusive interpretation of the Penn interpretation. (An approach similar to this was employed by Curien and Ghelli, although they tied coercion variables to particular type variables, and abstracted them automatically in polymorphic functions.)

This approach can be made to work, but the necessary facilities quickly become complicated as one scales the language to support additional features such as modules or higher-order type constructors. In this paper I propose a simpler approach in which coercion abstractions and variables will initially not be necessary at all. Although we will find coercion variables necessary to extend the technique to recursive types, even then the coercions employed at instances of subsumption will still be closed and no coercion abstractions will be necessary. The cost of this simplified target language will be a somewhat more complicated translation.

The translation is based on an interpretation of subtyping using intersection types that was first suggested by John Mitchell and explored further by Benjamin Pierce [9, Section 3.5.1]. In the Mitchell-Pierce interpretation, the bounded quantified type $\forall\alpha{\leq}\tau.\,\sigma(\alpha)$ is interpreted to mean $\forall\alpha.\,\sigma(\alpha \wedge \tau)$. In the former type, any type argument is required to be a subtype of the given bound τ; in its interpretation, any type argument is permitted, but is cut down to a subtype of the bound wherever it is used.

Pierce observed that this encoding does not entirely work, because it fails to validate the most general rule for subtyping of bounded quantified types. However, that failure turns out to be an artifact of the particular type theory, $F_\wedge$ [9, 10], that Pierce was using. I show that in a moderately more expressive type theory, the Mitchell-Pierce interpretation in fact becomes a valid encoding.

The Mitchell-Pierce encoding is useful for our purposes because it allows us to eliminate bounded quantification fully, without any need for coercion abstractions. Instead, subsumption coercions can be constructed entirely locally, even in the case of bounded quantification: the promotion of a type variable to its upper bound is implemented simply by the coercion from $\alpha \wedge \tau$ to τ.

Since we assume an inclusive interpretation of subtyping, where subsumption has no run-time action, the compilation process of this paper will make no changes to the type- and coercion-erasure of the program in question. The action of compilation is on the types, in reducing the high-level language feature of subtyping to lower-level static coercions. The resulting language, though larger (by the introduction of coercions), is simpler and enjoys entirely deterministic and syntax-directed type checking.

This paper is organized as follows: I begin by developing the translation eliminating subtyping in two steps. First, in Section 2, I present the translation in an $F_\wedge$-like type theory, making clear exactly what typing rules are necessary to validate the Mitchell-Pierce encoding. The target language for this version of the translation will still contain subtyping and will not enjoy tractable type checking, so it will not suffice for our ultimate purposes. Then, in Section 3, I reformalize the translation with a target language where explicit coercions replace subtyping and that is easily typechecked. In Section 4, I extend these results to account for recursive types. Finally, in Section 5, I give a semantics for the (second) target language that makes precise the notion that its coercions have no run-time effect. In what follows, familiarity is assumed with the polymorphic lambda calculus, subtyping, bounded quantification, and intersection types.

types	τ	$::=$	$\alpha \mid \mathtt{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid$ $\forall\alpha{\leq}\tau_1.\tau_2 \mid \mathtt{top}$
terms	e	$::=$	$x \mid i \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid (e_1, e_2) \mid$ $e.1 \mid e.2 \mid \Lambda\alpha{\leq}\tau.v \mid e[\tau]$
values	v	$::=$	$x \mid i \mid \lambda x{:}\tau.e \mid (v_1, v_2) \mid \Lambda\alpha{\leq}\tau.v$
contexts	Γ	$::=$	$\epsilon \mid \Gamma, \alpha{\leq}\tau \mid \Gamma, x{:}\tau$

Figure 1: Source Syntax

Judgement	Interpretation
$\vdash \Gamma$ ok	Γ is a valid context
$\Gamma \vdash \tau$ type	τ is a valid type
$\Gamma \vdash e : \tau$	e is a valid term of type τ
$\Gamma \vdash \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2

Figure 2: Source and (First) Target Judgements

2 The Mitchell-Pierce Interpretation

The source language for the translation is $F_\leq$ [5] augmented with products and a base type (int), the syntax for which is given in Figure 1, and the judgement forms for which are given in Figure 2. The typing and subtyping rules for the source language are standard; we discuss the most important rules below and the full system (for the language's final form) is summarized in Appendix A. Note the use of a value restriction in the syntax of type abstractions; this is to ensure that there are no problems in passing to a type-erasure semantics in Section 5. In what follows, we will write the simultaneous capture-avoiding substitution of $E_1, \ldots, E_n$ for $X_1, \ldots, X_n$ in E as $E[E_1 \cdots E_n / X_1 \cdots X_n]$. As usual, we will consider alpha-equivalent expressions to be identical.

The target language of the encoding is similar to the source, except that bounded quantification is replaced by simple quantification, and binary intersection types are added. The target syntax appears in Figure 3; the target's judgement forms are the same as for the source (Figure 2). The typing and subtyping rules for the target language are standard, except that we will add two somewhat unusual rules in Section 2.1 and we will have no need for the intersection type distributivity rules. The full system is summarized in Appendix A.1.

The idea to the Mitchell-Pierce interpretation is the bounded quantified type is defined in terms of ordinary quantification and intersection types:

$$\forall\alpha{\leq}\tau_1.\tau_2 \stackrel{\text{def}}{=} \forall\alpha.\,\tau_2[\alpha \wedge \tau_1/\alpha]$$

The left-hand type includes type abstractions that may be applied to any subtype of the given bound τ_1. The encoding relaxes this, allowing its members to be applied to any type, but then cuts that type down to a subtype of τ_1 wherever it is used.

When the type argument, say τ, is in fact a subtype of the bound—as will always be the case in target programs

types	τ	$::=$	$\alpha \mid \mathtt{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid$ $\forall\alpha.\tau \mid \tau_1 \wedge \tau_2 \mid \mathtt{top}$
terms	e	$::=$	$x \mid i \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid (e_1, e_2) \mid$ $e.1 \mid e.2 \mid \Lambda\alpha.v \mid e[\tau]$
values	v	$::=$	$x \mid i \mid \lambda x{:}\tau.e \mid (v_1, v_2) \mid \Lambda\alpha.v$
contexts	Γ	$::=$	$\epsilon \mid \Gamma, \alpha \mid \Gamma, x{:}\tau$

Figure 3: (First) Target Syntax

resulting from well-typed source programs—the types τ and $\tau \wedge \tau_1$ will be equivalent, and thus the result types $\tau_2[\tau/\alpha]$ and $\tau_2[\tau \wedge \tau_1/\alpha]$ will also be equivalent. This means that the application of a type abstraction works as expected. However, within the body of a type abstraction, $\alpha \wedge \tau_1$ can be shown to be a subtype of τ_1 without making any assumptions about α, and thus promotion of type variables to their upper bounds also works as expected.

We explore this in greater detail by considering three of the most important typing rules of the source language, the subtyping rule for type variables and the typing rules for type abstraction and application, and the images of those rules under the encoding.

- The subtyping rule for variables states that any type variables is a subtype of its given upper bound:

$$\frac{}{\Gamma \vdash \alpha \leq \tau} \; ((\alpha{\leq}\tau) \in \Gamma)$$

  The invariant of the encoding is that any type variable is replaced by the intersection of that variable with its upper bound, so when α has upper bound τ, it is everywhere replaced by $\alpha \wedge \tau$. Thus, the image of this rule's conclusion is $\Gamma \vdash \alpha \wedge \tau \leq \tau$, which certainly holds.[1]

- The typing rule for type abstractions is as follows:

$$\frac{\Gamma, \alpha{\leq}\tau \vdash v : \tau' \quad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \Lambda\alpha{\leq}\tau.v : \forall\alpha{\leq}\tau.\tau'} \; (\alpha \notin \mathrm{Dom}(\Gamma))$$

  The image of this rule's first antecedent is:

$$\Gamma, \alpha \vdash v[\alpha \wedge \tau/\alpha] : \tau'[\alpha \wedge \tau/\alpha]$$

  We may assume that this judgement holds and conclude, by the usual rule for type abstraction formation, that

$$\Gamma \vdash \Lambda\alpha.\, v[\alpha \wedge \tau/\alpha] : \forall\alpha.\, \tau'[\alpha \wedge \tau/\alpha]$$

  holds, which is the image of the rule's conclusion.

- The typing rule for type application is as follows:

$$\frac{\Gamma \vdash e : \forall\alpha{\leq}\tau_1.\tau_2 \quad \Gamma \vdash \tau \leq \tau_1}{\Gamma \vdash e[\tau] : \tau_2[\tau/\alpha]}$$

  The image of this rule's first antecedent is:

$$\Gamma \vdash e : \forall\alpha.\, \tau_2[\alpha \wedge \tau_1/\alpha]$$

  From this, using the usual rule for type application, we deduce that:

$$\Gamma \vdash e[\tau] : \tau_2[\tau \wedge \tau_1/\alpha]$$

  Certainly $\tau \wedge \tau_1 \leq \tau$, and, by the second antecedent, $\tau \leq \tau \wedge \tau_1$. Using the former subtyping relationship in positive positions of τ_2 and the latter in negative ones, we obtain

$$\Gamma \vdash \tau_2[\tau \wedge \tau_1/\alpha] \leq \tau_2[\tau/\alpha]$$

  and hence we may conclude by subsumption that

$$\Gamma \vdash e[\tau] : \tau_2[\tau/\alpha]$$

  holds, which is the image of the rule's conclusion.

[1]Strictly speaking, the image is $\Gamma' \vdash \alpha \wedge \tau' \leq \tau'$ where Γ' and τ' are the images of Γ and τ, but we will omit that level of detail in this informal discussion.

2.1 Quantifier subtyping

The preceding discussion shows that the encoding validates three of the four rules for bounded quantification, and it is easy to show that it also validates all the rules not relating to bounded quantification. However, a complication arises with the remaining rule, the subtyping rule for bounded quantified types:

$$\frac{\Gamma \vdash \tau_1' \leq \tau_1 \quad \Gamma, \alpha{\leq}\tau_1' \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash \forall\alpha{\leq}\tau_1.\tau_2 \;\leq\; \forall\alpha{\leq}\tau_1'.\tau_2'}$$

It is not so obvious that the encoding validates this rule. Consider the judgement:

$$\Gamma \vdash \forall\alpha{\leq}\mathtt{top}.\alpha \;\leq\; \forall\alpha{\leq}\mathtt{int}.\alpha$$

Certainly this judgement holds in the source language, since $\mathtt{int} \leq \mathtt{top}$. However, the image of this judgement under the encoding is:

$$\Gamma \vdash \forall\alpha.\, \alpha \wedge \mathtt{top} \leq \forall\alpha.\, \alpha \wedge \mathtt{int}$$

This judgement does not follow from the usual subtyping rule for (unbounded) quantified types, since $\alpha \wedge \mathtt{top} \not\leq \alpha \wedge \mathtt{int}$. In languages where the usual rule is the only rule for subtyping quantified types (such as Pierce's $F_\wedge$), the encoding fails.

One way to save the encoding is to restrict the source language by replacing the $F_\leq$ subtyping rule with the "Kernel Fun" rule, which requires the bounds τ_1 and τ_1' to be identical. With such a restriction in the source language, the problem does not arise.

However, there is no need to do this. We can also make the encoding work by strengthening the target language. We will strengthen the target language by adding subtyping rules that allow the implicit elimination and formation of quantified types:

$$\frac{\Gamma \vdash \forall\alpha.\tau \text{ type} \quad \Gamma \vdash \tau' \text{ type}}{\Gamma \vdash \forall\alpha.\tau \leq \tau[\tau'/\alpha]}$$

$$\frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \tau \leq \forall\alpha.\tau} \; (\alpha \text{ not free in } \tau)$$

The former rule allows quantified types to be instantiated implicitly using subtyping, rather than explicitly using the elimination construct for quantified types ($e[\tau]$). The latter rule similarly allows implicit formation of quantified types.

Note that although the former rule makes the usual elimination construct redundant, the latter rule cannot replace the formation construct $\Lambda\alpha.e$, because it does not provide any binding of α to be used in the body e. A stronger typing rule, as opposed to subtyping, can make the formation construct unnecessary, but for our purposes we will have no need for it.

These rules are semantically well-justified in a type-erasure setting, in which type abstraction and type application have no semantic effect. However, they make type-checking problematic, so they are rarely used in practical programming languages. Nevertheless, this language serves well to illustrate the Mitchell-Pierce interpretation. Moreover, with the elimination of subtyping in favor of explicit coercions in Section 3, our target language will be easily typechecked.

With the addition of these two rules, the encoding now validates the subtyping rule for quantified types. Recalling the example above:

$$\begin{array}{rcl} \forall\alpha.\,\alpha \wedge \mathtt{top} & \leq & \forall\alpha'.\,\forall\alpha.\,\alpha \wedge \mathtt{top} \\ & \leq & \forall\alpha'.\,(\alpha' \wedge \mathtt{int}) \wedge \mathtt{top} \\ & = & \forall\alpha.\,(\alpha \wedge \mathtt{int}) \wedge \mathtt{top} \\ & \leq & \forall\alpha.\,\alpha \wedge \mathtt{int} \end{array}$$

The first line follows by implicit formation, the second by implicit elimination (beneath the outermost quantifier) using $\alpha' \wedge \mathtt{int}$, the third by alpha conversion, and the last by the lower bound property of intersection types.

More generally, suppose the images of the antecedents of the subtyping rule hold, that is $\Gamma \vdash \tau_1' \leq \tau_1$ and

$$\Gamma, \alpha \vdash \tau_2[\alpha \wedge \tau_1'/\alpha] \leq \tau_2'[\alpha \wedge \tau_1'/\alpha]$$

First, observe that $(\alpha \wedge \tau_1') \wedge \tau_1 \leq \alpha \wedge \tau_1'$ and vice versa (for any type variable α). The shown direction follows from the lower bound property of intersection types, and the converse follows from the greatest lower bound property, since $\tau_1' \leq \tau_1$ is given by the first antecedent. It follows from this that

$$\Gamma, \alpha \vdash \tau_2[(\alpha \wedge \tau_1') \wedge \tau_1/\alpha] \leq \tau_2[\alpha \wedge \tau_1'/\alpha]$$

using the shown direction in positive positions of τ_2 and its converse in negative positions.

Now we may show that the rule's image holds, in an analogous manner to the example:

$$\begin{array}{rcl} \forall\alpha.\,\tau_2[\alpha \wedge \tau_1/\alpha] & \leq & \forall\alpha'.\,\forall\alpha.\,\tau_2[\alpha \wedge \tau_1/\alpha] \\ & \leq & \forall\alpha'.\,\tau_2[(\alpha' \wedge \tau_1') \wedge \tau_1/\alpha] \\ & = & \forall\alpha.\,\tau_2[(\alpha \wedge \tau_1') \wedge \tau_1/\alpha] \\ & \leq & \forall\alpha.\,\tau_2[\alpha \wedge \tau_1'/\alpha] \\ & \leq & \forall\alpha.\,\tau_2'[\alpha \wedge \tau_1'/\alpha] \end{array}$$

The first line follows by implicit formation, the second by implicit elimination using $\alpha' \wedge \tau_1'$, the third by alpha conversion, the fourth by the fact shown above, and the last by the second antecedent's image.

2.2 Formalization

The encoding is formalized as a syntax-directed type translation $|\tau|_\Gamma$, term translation $|e|_\Gamma$, and context translation $|\Gamma|$. We begin with the type translation (shown in Figure 4), which we define in two parts. The first part is a parametric translation $|\cdot|$, which does not modify type variables. The key clause is the one for quantified types, which states:

$$|\forall\alpha{\leq}\tau_1.\tau_2| \stackrel{\text{def}}{=} \forall\alpha.|\tau_2|[\alpha \wedge |\tau_1|/\alpha]$$

$$\begin{array}{rcl} |\alpha| & \stackrel{\text{def}}{=} & \alpha \\ |\mathtt{int}| & \stackrel{\text{def}}{=} & \mathtt{int} \\ |\tau_1 \rightarrow \tau_2| & \stackrel{\text{def}}{=} & |\tau_1| \rightarrow |\tau_2| \\ |\tau_1 \times \tau_2| & \stackrel{\text{def}}{=} & |\tau_1| \times |\tau_2| \\ |\forall\alpha{\leq}\tau_1.\tau_2| & \stackrel{\text{def}}{=} & \forall\alpha.|\tau_2|[\alpha \wedge |\tau_1|/\alpha] \\ |\mathtt{top}| & \stackrel{\text{def}}{=} & \mathtt{top} \\ \\ \mathit{Sub}(\epsilon;\tau) & \stackrel{\text{def}}{=} & \tau \\ \mathit{Sub}((\Gamma,\alpha{\leq}\tau');\tau) & \stackrel{\text{def}}{=} & \mathit{Sub}(\Gamma;\tau[\alpha \wedge |\tau'|/\alpha]) \\ \mathit{Sub}((\Gamma,x{:}\tau');\tau) & \stackrel{\text{def}}{=} & \mathit{Sub}(\Gamma;\tau) \\ |\tau|_\Gamma & \stackrel{\text{def}}{=} & \mathit{Sub}(\Gamma;|\tau|) \end{array}$$

Figure 4: Type Translation

$$\begin{array}{rcl} |x|_\Gamma & \stackrel{\text{def}}{=} & x \\ |i|_\Gamma & \stackrel{\text{def}}{=} & i \\ |\lambda x{:}\tau.e|_\Gamma & \stackrel{\text{def}}{=} & \lambda x{:}|\tau|_\Gamma.|e|_\Gamma \\ |e_1 e_2|_\Gamma & \stackrel{\text{def}}{=} & (|e_1|_\Gamma)(|e_2|_\Gamma) \\ |(e_1,e_2)|_\Gamma & \stackrel{\text{def}}{=} & (|e_1|_\Gamma, |e_2|_\Gamma) \\ |e.i|_\Gamma & \stackrel{\text{def}}{=} & (|e|_\Gamma).i \\ |\Lambda\alpha{\leq}\tau.e|_\Gamma & \stackrel{\text{def}}{=} & \Lambda\alpha.|e|_{(\Gamma,\alpha\leq\tau)} \\ |e[\tau]|_\Gamma & \stackrel{\text{def}}{=} & (|e|_\Gamma)[|\tau|_\Gamma] \end{array}$$

Figure 5: (First) Term Translation

The parametric type translation accounts for the upper bounds of all bound variables, but does not account for the upper bounds of free variables. Those are obtained by reference to the context. Thus, the second part is a context sensitive translation $|\cdot|_\Gamma$, which modifies type variables appropriately:

$$\begin{array}{rcl} \mathit{Sub}(\epsilon;\tau) & \stackrel{\text{def}}{=} & \tau \\ \mathit{Sub}((\Gamma,\alpha{\leq}\tau');\tau) & \stackrel{\text{def}}{=} & \mathit{Sub}(\Gamma;\tau[\alpha \wedge |\tau'|/\alpha]) \\ \mathit{Sub}((\Gamma,x{:}\tau');\tau) & \stackrel{\text{def}}{=} & \mathit{Sub}(\Gamma;\tau) \\ |\tau|_\Gamma & \stackrel{\text{def}}{=} & \mathit{Sub}(\Gamma;|\tau|) \end{array}$$

The term translation (Figure 5) and context translation (Figure 6) simply apply the appropriate translation to their component types and delete upper bounds from variables.

With this formalization, we can state the following static correctness theorem, which summarizes the informal discussion above. We distinguish between judgements in the source and target languages by marking the turnstiles $\vdash_S$ or $\vdash_T$, respectively.

Theorem 2.1

1. *If* $\Gamma \vdash_S \tau$ type *and* $\vdash_S \Gamma$ ok *then* $|\Gamma| \vdash_T |\tau|_\Gamma$ type.
2. *If* $\Gamma \vdash_S \tau_1 \leq \tau_2$ *and* $\vdash_S \Gamma$ ok *then* $|\Gamma| \vdash_T |\tau_1|_\Gamma \leq |\tau_2|_\Gamma$.
3. *If* $\Gamma \vdash_S e : \tau$ *and* $\vdash_S \Gamma$ ok *then* $|\Gamma| \vdash_T |e|_\Gamma : |\tau|_\Gamma$.

$$\begin{aligned} |\epsilon| &\stackrel{\text{def}}{=} \epsilon \\ |\Gamma, \alpha{\le}\tau| &\stackrel{\text{def}}{=} |\Gamma|, \alpha \\ |\Gamma, x{:}\tau| &\stackrel{\text{def}}{=} |\Gamma|, x{:}|\tau|_\Gamma \end{aligned}$$

Figure 6: Context Translation

types	τ	::=	$\alpha \mid \texttt{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \forall\alpha.\tau \mid \tau_1 \wedge \tau_2 \mid \texttt{top}$
terms	e	::=	$x \mid i \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid (e_1, e_2) \mid e.i \mid \Lambda\alpha.v \mid c\, e$
coercions	c	::=	$\texttt{id} \mid c_1 \circ c_2 \mid c_1 \to c_2 \mid c_1 \times c_2 \mid \forall\alpha.c \mid \langle c_1, c_2\rangle \mid \pi_i[\tau_1 \wedge \tau_2] \mid \texttt{top}[\tau] \mid \texttt{app}[\forall\alpha.\tau]\,\tau' \mid \texttt{gen}$
contexts	Γ	::=	$\epsilon \mid \Gamma, \alpha \mid \Gamma, x{:}\tau$
values	v	::=	$x \mid i \mid \lambda x{:}\tau.e \mid (v_1, v_2) \mid \Lambda\alpha.v \mid c\, v$

Figure 7: Coercion Calculus Syntax

Although I do not formalize an operational semantics for the source or target language here, it is easy to see that the encoding is dynamically correct in any semantics that respects type erasure, since any source term's erasure is identical to its translation's erasure. This observation also neatly addresses the issue of the translation's coherence: any two translations of a term must be equivalent, since each is equivalent to the source term.

3 The Coercion Interpretation

Our end goal is to eliminate subtyping entirely, not just to eliminate bounded quantification in favor of intersection types. To that end, we define a coercion-based calculus to serve as a target language, and a translation from the original source language to the new coercion language. In the coercion calculus, "subtyping" relationships will be represented by explicit coercions, which will make typechecking easy.

The syntax of the coercion calculus is given in Figure 7. Aside from the new syntactic class of coercions, the syntax is similar to the target language from Section 2. To aid in typechecking, several coercion constructs (π, `top`, and `app`) include type annotations indicating their domains; in discussion we will omit these annotations when they are clear from context.

The judgements of the coercion calculus are given in Figure 8. The judgements for type formation and typing of terms are standard. The final judgement, for typing coercions, is the analog of the subtyping judgement; $\Gamma \vdash c : \tau_1 \Rightarrow \tau_2$ indicates that c is a coercion from τ_1 to τ_2. In this case, the coercion c may be thought of as a *witness* that τ_1 is a subtype of τ_2. When judgements in the coercion calculus must be distinguished from judgements in the source language, we will do so by marking the turnstile $\vdash_C$.

Most of the terms of the coercion calculus have their usual meanings. The new term construct, $c\,e$, indicates the application of a coercion to a term; this may be thought of as syntactically indicating the use of subsumption. Also

Judgement	Interpretation
$\vdash \Gamma$ ok	Γ is a valid context
$\Gamma \vdash \tau$ type	τ is a valid type
$\Gamma \vdash e : \tau$	e is a valid term of type τ
$\Gamma \vdash c : \tau_1 \Rightarrow \tau_2$	c is a valid coercion from τ_1 to τ_2

Figure 8: Coercion Calculus Judgements

note that the term construct for type application is omitted; that construct is replaced by the `app` coercion.

The coercion constructs are interpreted as follows:

- The coercions `id` and $c_1 \circ c_2$ denote identity and composition. They may be thought of as witnesses to the reflexivity and transitivity subtyping rules.
- The coercions $c_1 \to c_2$, $c_1 \times c_2$, and $\forall\alpha.c$ lift coercions over the basic type operators. For example, $c_1 \to c_2$ modifies a function by applying c_1 to its argument and c_2 to its result. These constructs are witnesses to the subtyping rules for compatibility with the basic type operators.
- The coercion $\langle c_1, c_2\rangle$ is the introduction construct for intersection types, and intuitively works by applying each of the two given coercions and collecting the results. It has the typing rule:

  $$\frac{\Gamma \vdash c_1 : \tau \Rightarrow \tau_1 \quad \Gamma \vdash c_2 : \tau \Rightarrow \tau_2}{\Gamma \vdash \langle c_1, c_2\rangle : \tau \Rightarrow \tau_1 \wedge \tau_2}$$

  This coercion is the witness to the greatest lower bound rule for intersection types.
- The coercion π_i is the elimination construct for intersection types, and works by selecting one of the two results from an intersection introduction. It has the typing rule:

  $$\frac{\Gamma \vdash \tau_1 \wedge \tau_2 \text{ type}}{\Gamma \vdash \pi_i[\tau_1 \wedge \tau_2] : \tau_1 \wedge \tau_2 \Rightarrow \tau_i} \quad (i = 1, 2)$$

  This is the witness to the lower bound rule for intersection types.

  Note that using the last two coercions we can define a compatibility coercion for intersection types:

  $$c_1 \wedge c_2 \stackrel{\text{def}}{=} \langle c_1 \circ \pi_1, c_2 \circ \pi_2\rangle$$
- The coercion `top` is the introduction construct for the type of the same name and witnesses the subtyping top rule.
- The application coercion $\texttt{app}\,\tau$ is the elimination construct for quantified types. It witnesses the implicit elimination rule from Section 2.1 and has the typing rule:

  $$\frac{\Gamma \vdash \forall\alpha.\tau \text{ type} \quad \Gamma \vdash \tau' \text{ type}}{\Gamma \vdash \texttt{app}[\forall\alpha.\tau]\,\tau' : \forall\alpha.\tau \Rightarrow \tau[\tau'/\alpha]}$$

  Note that the usual elimination form for quantified types can be built from this coercion:

  $$e[\tau] \stackrel{\text{def}}{=} (\texttt{app}\,\tau)\,e$$

- The generalization coercion `gen` introduces a member of a quantified type by wrapping a type abstraction (*i.e.*, $\Lambda\alpha.-$) around its argument. It witnesses the implicit introduction rule from Section 2.1 and has the typing rule:

$$\frac{\Gamma \vdash \tau \ \text{type}}{\Gamma \vdash \mathtt{gen} : \tau \Rightarrow \forall\alpha.\tau} \ (\alpha \text{ not free in } \tau)$$

The typing rules for coercions and the rest of the coercion calculus are summarized in Appendix B.1. Typechecking for this language is easy, due to a unique typing property for coercions:

Proposition 3.1 *Suppose Γ, c, and τ are given. Then there exists at most one τ' such that $\Gamma \vdash c : \tau \Rightarrow \tau'$ and, conversely, there exists at most one τ' such that $\Gamma \vdash c : \tau' \Rightarrow \tau$.*

Providing this property is the purpose of the domain annotations on the π, `top`, and `app` coercions. Without them, $\mathtt{top} \to \mathtt{id}$ does not have a unique codomain, for example.

3.1 The Coercion Translation

With a target language in place, we can now define the translation eliminating subtyping. The type and context translations are identical to those used in the first translation (Figures 4 and 6). The term translation is different, of course, since it must now provide coercion expressions. Furthermore, since the necessary coercion expressions are determined by typing and subtyping derivations (not by the syntax of the terms themselves), the translation is given as a type-directed translation.

The type-directed translation is given as a term translation and a subtyping translation. The term translation is given by a judgement $\Gamma \vdash_{SC} e : \tau \ \rightrightarrows\ e'$, meaning that e has type τ (in the source) and e' is its translation. The subtyping translation is given by a judgement $\Gamma \vdash_{SC} \tau_1 \leq \tau_2 \ \rightrightarrows\ c$, meaning that the coercion c witnesses that (in the source) τ_1 is a subtype of τ_2. As usual, rules in the term translation are in one-to-one correspondence with source typing rules, and rules in the subtyping translation with the source subtyping rules.

The interesting rules are the ones that deal with quantified types. We proceed by looking carefully at these rules; the complete rules to the translation appear in Appendix C.

- The translation rule for variables is:

$$\frac{}{\Gamma \vdash \alpha \leq \tau \ \rightrightarrows\ \pi_2} \ ((\alpha{\leq}\tau) \in \Gamma)$$

The correctness criterion for subtyping translations is that if

$$\Gamma \vdash_{SC} \tau_1 \leq \tau_2 \ \rightrightarrows\ c$$

(and Γ is well-formed) then

$$|\Gamma| \vdash_C c : |\tau_1|_\Gamma \Rightarrow |\tau_2|_\Gamma$$

This rule establishes that criterion since $|\alpha|_\Gamma = \alpha \wedge |\tau|_\Gamma$ and π_2 takes $\alpha \wedge |\tau|_\Gamma$ to $|\tau|_\Gamma$.

- The translation rule for type abstractions is simply:

$$\frac{\begin{array}{c}\Gamma \vdash_S \tau \ \text{type} \\ \Gamma, \alpha{\leq}\tau \vdash_{SC} v : \tau' \ \rightrightarrows\ v'\end{array}}{\Gamma \vdash_{SC} \Lambda\alpha{\leq}\tau.v : \forall\alpha{\leq}\tau.\tau' \ \rightrightarrows\ \Lambda\alpha.v'} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\begin{array}{rcl}
\mathtt{map}[\alpha.\alpha](c_+, c_-) & \overset{\text{def}}{=} & c_+ \\
\mathtt{map}[\alpha.\beta](c_+, c_-) & \overset{\text{def}}{=} & \mathtt{id} \quad (\text{for } \alpha \neq \beta) \\
\mathtt{map}[\alpha.\mathtt{int}](c_+, c_-) & \overset{\text{def}}{=} & \mathtt{id} \\
\mathtt{map}[\alpha.\,\tau_1 \to \tau_2](c_+, c_-) & \overset{\text{def}}{=} & \mathtt{map}[\alpha.\tau_1](c_-, c_+) \to \\
 & & \mathtt{map}[\alpha.\tau_2](c_+, c_-) \\
\mathtt{map}[\alpha.\,\tau_1 \times \tau_2](c_+, c_-) & \overset{\text{def}}{=} & \mathtt{map}[\alpha.\tau_1](c_+, c_-) \times \\
 & & \mathtt{map}[\alpha.\tau_2](c_+, c_-) \\
\mathtt{map}[\alpha.\,(\forall\beta.\tau)](c_+, c_-) & \overset{\text{def}}{=} & \forall\beta.\,\mathtt{map}[\alpha.\tau](c_+, c_-) \\
 & & (\text{where } \beta \text{ is fresh}) \\
\mathtt{map}[\alpha.\,\tau_1 \wedge \tau_2](c_+, c_-) & \overset{\text{def}}{=} & \mathtt{map}[\alpha.\tau_1](c_+, c_-) \wedge \\
 & & \mathtt{map}[\alpha.\tau_2](c_+, c_-) \\
\mathtt{map}[\alpha.\mathtt{top}](c_+, c_-) & \overset{\text{def}}{=} & \mathtt{id}
\end{array}$$

Figure 9: Definition of `map`

The correctness criterion for term translations is the usual, that if $\Gamma \vdash_{SC} e : \tau \ \rightrightarrows\ e'$ (and Γ is well-formed) then $|\Gamma| \vdash_C e' : |\tau|_\Gamma$. This rule preserves that criterion since $\Lambda\alpha.v'$ has type

$$\begin{array}{rcl}
\forall\alpha.|\tau'|_{(\Gamma,\alpha\leq\tau)} & = & \forall\alpha.\,\mathit{Sub}(\Gamma; |\tau'|[\alpha \wedge |\tau|/\alpha]) \\
 & = & \mathit{Sub}(\Gamma; \forall\alpha.\,|\tau'|[\alpha \wedge |\tau|/\alpha]) \\
 & = & |\forall\alpha{\leq}\tau.\tau'|_\Gamma
\end{array}$$

The second line follows since α is not in the domain of Γ, the other two are direct from the definitions.

- The translation rule for type application is

$$\frac{\Gamma \vdash_{SC} e : \forall\alpha{\leq}\tau_1.\tau_2 \ \rightrightarrows\ e' \quad \Gamma \vdash_{SC} \tau \leq \tau_1 \ \rightrightarrows\ c}{\begin{array}{l}\Gamma \vdash_{SC} e[\tau] : \tau_2[\tau/\alpha] \ \rightrightarrows \\ \qquad (\mathtt{map}[\alpha.|\tau_2|_\Gamma](\pi_1, \langle \mathtt{id}, c\rangle) \circ \mathtt{app}\ |\tau|_\Gamma)\, e'\end{array}} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

where $\mathtt{map}[\alpha.\tau](c_+, c_-)$ applies c_+ at all positive occurrences of α in τ and c_- at all negative occurrences. Its definition is given in Figure 9 and its typing behavior is specified by the following lemma:

Lemma 3.2 *If* $\Gamma, \alpha \vdash \tau$ type *and* $\Gamma \vdash c_+ : \tau_1 \Rightarrow \tau_2$ *and* $\Gamma \vdash c_- : \tau_2 \Rightarrow \tau_1$ *and* $\vdash \Gamma, \alpha$ ok *then* $\Gamma \vdash \mathtt{map}[\alpha.\tau](c_+, c_-) : \tau[\tau_1/\alpha] \Rightarrow \tau[\tau_2/\alpha]$.

Since π_1 takes $|\tau|_\Gamma \wedge |\tau_1|_\Gamma$ to $|\tau|_\Gamma$, and $\langle \mathtt{id}, c\rangle$ goes the opposite direction, using Lemma 3.2, we may deduce that the map expression above takes $|\tau_2|_\Gamma[|\tau|_\Gamma \wedge |\tau_1|_\Gamma/\alpha]$ to $|\tau_2|_\Gamma[|\tau|_\Gamma/\alpha]$.

When composed with $\mathtt{app}\ |\tau|_\Gamma$, the resulting coercion takes $|\forall\alpha{\leq}\tau_1.\tau_2|_\Gamma = \forall\alpha.|\tau_2|_\Gamma[\alpha \wedge |\tau_1|_\Gamma/\alpha]$ to $|\tau_2|_\Gamma[|\tau|_\Gamma/\alpha]$. Using an easy-to-show substitution lemma, the latter is equal to $|\tau_2[\tau/\alpha]|_\Gamma$, as required.

- Finally, the translation rule for subtyping of quantified types is:

$$\frac{\Gamma \vdash_{SC} \tau_1' \leq \tau_1 \ \rightrightarrows\ c_1 \quad \Gamma, \alpha{\leq}\tau_1' \vdash_{SC} \tau_2 \leq \tau_2' \ \rightrightarrows\ c_2}{\begin{array}{l}\Gamma \vdash_{SC} \forall\alpha{\leq}\tau_1.\tau_2 \leq \forall\alpha{\leq}\tau_1'.\tau_2' \ \rightrightarrows \\ \qquad \forall\alpha.(c_2 \circ \\ \qquad\qquad \mathtt{map}[\alpha.|\tau_2|_\Gamma](\pi_1, \langle \mathtt{id}, c_1 \circ \pi_2\rangle) \circ \\ \qquad\qquad \mathtt{app}\ (\alpha \wedge |\tau_1'|_\Gamma)) \circ \\ \qquad \mathtt{gen}\end{array}} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

Let us examine this coercion starting at the middle. Let α be a type variable. Then π_1 takes $(\alpha \wedge |\tau_1'|_\Gamma) \wedge |\tau_1|_\Gamma$ to $\alpha \wedge |\tau_1'|_\Gamma$; and $\langle \mathtt{id}, c_1 \circ \pi_2 \rangle$ goes the other direction, since $c_1 \circ \pi_2$ takes $\alpha \wedge |\tau_1'|_\Gamma$ to $|\tau_1|_\Gamma$. Thus the `map` expression takes $|\tau_2|_\Gamma[(\alpha \wedge |\tau_1'|_\Gamma) \wedge |\tau_1|_\Gamma/\alpha]$ to $|\tau_2|_\Gamma[\alpha \wedge |\tau_1'|_\Gamma/\alpha]$.

When composed on each end with c_2 and $\mathtt{app}(\alpha \wedge |\tau_1'|_\Gamma)$, the result takes $\forall\alpha'.|\tau_2|_\Gamma[\alpha' \wedge |\tau_1|_\Gamma/\alpha]$ to $|\tau_2'|_\Gamma[\alpha \wedge |\tau_1'|_\Gamma/\alpha]$. Thus, discharging the variable α, the entire $\forall$ expression takes $\forall\alpha.\forall\alpha'.|\tau_2|_\Gamma[\alpha' \wedge |\tau_1|_\Gamma/\alpha]$ to $|\forall\alpha{\le}\tau_1'.\tau_2'|_\Gamma$. When composed with `gen` (and employing a change of variables), the result takes $|\forall\alpha{\le}\tau_1.\tau_2|_\Gamma$ to $|\forall\alpha{\le}\tau_1'.\tau_2'|_\Gamma$, as required.

The static correctness of this translation is formalized by the following theorem, which summarizes the informal discussion above:

Theorem 3.3

1. *If* $\Gamma \vdash_S \tau$ type *and* $\vdash_S \Gamma$ ok *then* $|\Gamma| \vdash_T |\tau|_\Gamma$ type.
2. *If* $\Gamma \vdash_{SC} \tau_1 \le \tau_2 \;\rightrightarrows\; c$ *and* $\vdash_S \Gamma$ ok *then* $|\Gamma| \vdash_C c : |\tau_1|_\Gamma \Rightarrow |\tau_2|_\Gamma$.
3. *If* $\Gamma \vdash_{SC} e : \tau \;\rightrightarrows\; e'$ *and* $\vdash_S \Gamma$ ok *then* $|\Gamma| \vdash_T e' : |\tau|_\Gamma$.

As before, the translation can easily be seen to be dynamically correct in any semantics respecting type erasure, since any source term's erasure is identical to its translation's erasure.

4 Recursive Types

The translation above accounts for a basic source language supporting functions, products, and bounded polymorphism. A natural question then is whether the approach scales to larger, more expressive type systems. In fact, the translation above generalizes easily to account for source-level intersection and sum types, and dualizes nicely for bounded existential and union types (with appropriate enhancements to the target language). These extensions are omitted here because they add little to the present discussion. The approach also generalizes to support recursive types, but that extension is a bit more involved and merits some discussion.

Accounting for recursive types requires support for two subtyping principles. First, we must support the usual rule for subtyping recursive types:

$$\frac{\begin{array}{l}\Gamma, \alpha{\le}\mathtt{top} \vdash_S \tau \text{ type} \\ \Gamma, \alpha'{\le}\mathtt{top} \vdash_S \tau' \text{ type} \\ \Gamma, \alpha'{\le}\mathtt{top}, \alpha{\le}\alpha' \vdash_S \tau \le \tau'\end{array}}{\Gamma \vdash_S \mu\alpha.\tau \le \mu\alpha'.\tau'} \quad (\alpha, \alpha' \notin \mathrm{Dom}(\Gamma))$$

Second, we must be able to map a type isomorphism through any type expression (where we think of τ_1 and τ_2 as isomorphic when $\tau_1 \le \tau_2$ and $\tau_2 \le \tau_1$). This requirement arose both in type application and in subtyping of quantified types. Previously this was possible using only the usual subtyping rules, but this is not the case in the presence of recursive types.

The problem is that in the subtyping rule for recursive types above, the premise $\alpha \le \alpha'$ is useful only in positive positions; in negative positions the premise is oriented the wrong way. Consequently, we can map an isomorphism only through recursive types in which the recursive variables appears only positively. Nevertheless, we should have both orientations available since we are dealing with isomorphism and not merely subtyping.

The needed subtyping rule provides the usual premise for positive positions, but also provides the opposite premise for negative positions. To justify these premises the rule must ensure that the two types are in fact isomorphic, by requiring subtyping in each direction, as shown in the rule in Figure 10.

In the usual use of this rule, the appearances of α_1 in τ_1 are divided up into positive and negative appearances, with the negative ones marked α_1' in τ_1^+ and the positive ones in τ_1^-. This ensures that the recursive relationship is available in positive positions when showing left-to-right subtyping (as in the simple rule), and is available in negative positions when showing right-to-left subtyping, as required. Appearances of α_2 in τ_2 are similarly divided.

The necessary extensions to the syntax of the coercion calculus, principally new coercion forms, are given in Figure 11. Two new coercions (`fold` and `unfold`) are used to introduce and eliminate recursive types. The syntax for these coercions give their codomain and domain types, respectively, in order to preserve unique typing (Proposition 3.1).

More interesting are the coercions witnessing the subtyping and isomorphism rules for recursive types. The subtyping coercion, $\mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c)$, applies c to the body of a member of a recursive type, where c may call itself recursively through the coercion variable χ. The typing rule for `rec` is:

$$\frac{\Gamma, \alpha \vdash \tau \text{ type} \qquad \Gamma, \alpha' \vdash \tau' \text{ type} \qquad (\Gamma, \alpha, \alpha'); (\Phi, \chi : \alpha \Rightarrow \alpha') \vdash c : \tau \Rightarrow \tau'}{\Gamma; \Phi \vdash \mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c) : \mu\alpha.\tau \Rightarrow \mu\alpha'.\tau'} \quad (\alpha, \alpha' \notin \mathrm{Dom}(\Gamma), \chi \notin \mathrm{Dom}(\Phi))$$

Thus c uses χ to coerce subcomponents from type α to α'. Note that the introduction of coercion variables mandates the use of coercion contexts (Φ). However, it also is important to note that coercion contexts are necessary *only* for typing coercions; no coercion variable need ever appear free in a term.

The isomorphism coercion, $\mathtt{isorec}(\chi_+ : \alpha_1 \Rightarrow \alpha_2', \chi_- : \alpha_2 \Rightarrow \alpha_1'. c_+, c_-)$, applies c_+ to the body of a member of a recursive type, but simultaneously defines a reverse coercion c_-, and makes both coercions recursively available to each other under the names χ_+ and χ_- (respectively). The resulting typing rule is given in Figure 12.

As in the rule it witnesses, the `isorec` rule divides up the appearances of the recursive variables in the two types.

types	τ	$::=$	$\cdots \mid \mu\alpha.\tau$
coercions	c	$::=$	$\cdots \mid \chi \mid$
			$\mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c) \mid$
			$\mathtt{isorec}(\chi_+ : \alpha_1 \Rightarrow \alpha_2',$ $\chi_- : \alpha_2 \Rightarrow \alpha_1'.$ $c_+, c_-) \mid$
			$\mathtt{fold}[\mu\alpha.\tau] \mid \mathtt{unfold}[\mu\alpha.\tau]$
coercion contexts	Φ	$::=$	$\epsilon \mid \Phi, \chi : \tau_1 \Rightarrow \tau_2$

Figure 11: Extensions for Recursive Types

$$\frac{\begin{array}{c}\Gamma, \alpha_1{\le}\mathtt{top}, \alpha'_1{\le}\mathtt{top} \vdash_S \tau_1^+ \text{ type} \qquad \Gamma, \alpha_1{\le}\mathtt{top}, \alpha'_1{\le}\mathtt{top} \vdash_S \tau_1^- \text{ type} \\ \Gamma, \alpha_2{\le}\mathtt{top}, \alpha'_2{\le}\mathtt{top} \vdash_S \tau_2^+ \text{ type} \qquad \Gamma, \alpha_2{\le}\mathtt{top}, \alpha'_2{\le}\mathtt{top} \vdash_S \tau_2^- \text{ type} \\ \Gamma, \alpha'_1{\le}\mathtt{top}, \alpha'_2{\le}\mathtt{top}, \alpha_1{\le}\alpha'_2, \alpha_2{\le}\alpha'_1 \vdash_S \tau_1^+ \le \tau_2^+ \\ \Gamma, \alpha'_1{\le}\mathtt{top}, \alpha'_2{\le}\mathtt{top}, \alpha_1{\le}\alpha'_2, \alpha_2{\le}\alpha'_1 \vdash_S \tau_2^- \le \tau_1^- \end{array}}{\Gamma \vdash_S \mu\alpha_1.\tau_1 \le \mu\alpha_2.\tau_2} \quad \left(\begin{array}{l}\alpha_1, \alpha'_1, \alpha_2, \alpha'_2 \notin \mathrm{Dom}(\Gamma) \\ \tau_1 = \tau_1^+[\alpha_1/\alpha'_1] = \tau_1^-[\alpha_1/\alpha'_1] \\ \tau_2 = \tau_2^+[\alpha_2/\alpha'_2] = \tau_2^-[\alpha_2/\alpha'_2]\end{array}\right)$$

Figure 10: Recursive Type Isomorphism Rule

$$\frac{\begin{array}{c}\Gamma, \alpha_1, \alpha'_1 \vdash \tau_1^+ \text{ type} \qquad \Gamma, \alpha_1, \alpha'_1 \vdash \tau_1^- \text{ type} \\ \Gamma, \alpha_2, \alpha'_2 \vdash \tau_2^+ \text{ type} \qquad \Gamma, \alpha_2, \alpha'_2 \vdash \tau_2^- \text{ type} \\ \Gamma'; \Phi' \vdash c_+ : \tau_1^+ \Rightarrow \tau_2^+ \quad \Gamma'; \Phi' \vdash c_- : \tau_2^- \Rightarrow \tau_1^- \end{array}}{\Gamma; \Phi \vdash \mathtt{isorec}(\chi_+ : \alpha_1 \Rightarrow \alpha'_2, \chi_- : \alpha_2 \Rightarrow \alpha'_1.\, c_+, c_-) : \mu\alpha_1.\tau_1 \Rightarrow \mu\alpha_2.\tau_2} \quad \left(\begin{array}{l}\Gamma' = \Gamma, \alpha_1, \alpha'_1, \alpha_2, \alpha'_2 \\ \Phi' = \Phi, \chi_+ : \alpha_1 \Rightarrow \alpha'_2, \chi_- : \alpha_2 \Rightarrow \alpha'_1 \\ \alpha_1, \alpha'_1, \alpha_2, \alpha'_2 \notin \mathrm{Dom}(\Gamma), \chi_+, \chi_- \notin \mathrm{Dom}(\Phi) \\ \alpha_1 \text{ positive, } \alpha'_1 \text{ negative in } \tau_1^+ \\ \alpha_2 \text{ negative, } \alpha'_2 \text{ positive in } \tau_2^+ \\ \alpha_1 \text{ negative, } \alpha'_1 \text{ positive in } \tau_1^- \\ \alpha_2 \text{ positive, } \alpha'_2 \text{ negative in } \tau_2^- \\ \tau_1 = \tau_1^+[\alpha_1/\alpha'_1] = \tau_1^-[\alpha_1/\alpha'_1] \\ \tau_2 = \tau_2^+[\alpha_2/\alpha'_2] = \tau_2^-[\alpha_2/\alpha'_2]\end{array}\right)$$

Figure 12: Isomorphism Coercion Rule

In this version of the rule, the division into positive and negative is required to be the usual one. The requirement is imposed so that typechecking of coercions remains syntax directed (otherwise τ_1^+, τ_2^+, etc. are not determined by τ_1 and τ_2), ensuring easy typechecking. It is not required for type safety, so it could be relaxed, but doing so would be unlikely to provide any useful expressive power.

Fold and unfold operations in the source language are translated using `fold` and `unfold` coercions in the obvious manner. Subtyping of recursive types is translated by the rule:

$$\frac{\begin{array}{l}\Gamma, \alpha{\le}\mathtt{top} \vdash_S \tau \text{ type} \\ \Gamma, \alpha'{\le}\mathtt{top} \vdash_S \tau' \text{ type} \\ \Gamma, \alpha'{\le}\mathtt{top}, \alpha{\le}\alpha' \vdash_{SC} \tau \le \tau' \;\rightrightarrows\; c\end{array}}{\begin{array}{l}\Gamma \vdash_{SC} \mu\alpha.\tau \le \mu\alpha'.\tau' \;\rightrightarrows \\ \qquad \mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.\, \mathtt{map}[\alpha'.\tau'](\pi_1, \langle \mathtt{id}, \mathtt{top}\rangle) \circ c \circ \\ \qquad\qquad \mathtt{map}[\alpha.\tau](\langle \mathtt{id}, \langle \chi, \mathtt{top}\rangle\rangle, \pi_1))\end{array}}$$

This rule is made somewhat messy by the interpretation of α' as $\alpha' \wedge \mathtt{top}$; in practice, a compiler would optimize the case when an upper bound is `top`. The coercion's body first coerces τ to $\tau[\alpha \wedge (\alpha' \wedge \mathtt{top})/\alpha]$, thereby setting it up for c, after which it coerces the resulting $\tau'[\alpha' \wedge \mathtt{top}/\alpha']$ back down to τ'.

Finally we can define the necessary final clause of `map` using the isomorphism coercion by:[2]

$$\begin{array}{l} \qquad\qquad \mathtt{map}[\alpha.\,(\mu\beta.\tau)](c_+, c_-) \\ \qquad\qquad\qquad \stackrel{\text{def}}{=} \\ \mathtt{isorec}(\chi_+ : \beta_1 \Rightarrow \beta'_2, \chi_- : \beta_2 \Rightarrow \beta'_1. \\ \qquad \mathtt{map}[\beta.\tau](\chi_+, \chi_-) \circ \mathtt{map}[\alpha.\tau](c_+, c_-), \\ \qquad \mathtt{map}[\alpha.\tau](c_-, c_+) \circ \mathtt{map}[\beta.\tau](\chi_-, \chi_+)) \\ \text{(where } \beta_1, \beta'_1, \beta_2, \beta'_2, \chi_+, \chi_- \text{ are fresh)}\end{array}$$

[2]With this clause, some strengthening of the induction hypothesis in necessary to show Lemma 3.2, since χ_+ and χ_- do not operate on the same types.

5 Dynamic Semantics of the Coercion Calculus

It remains to put this compilation strategy on a solid footing by establishing an operational semantics for the coercion calculus. We desire two properties of the semantics: we want the usual type safety property, of course, but we also want to make explicit the inclusive nature of subtyping, that is, that coercions have no run-time effect.

We take the view that the run-time substance of a term is reflected in the term's *erasure*, the portion of the term remaining after all types, type abstractions and coercions are erased. Types, type abstractions and coercions will be viewed as having purely static importance.[3]

This view immediately advises the design of the operational semantics. Consider evaluation of the term $\langle c_1, c_2 \rangle v$. A naive approach would be to include a term form for intersection types (written, say, $\langle e_1, e_2 \rangle$), and to define the semantics so that $\langle c_1, c_2 \rangle v$ evaluates to $\langle c_1 v, c_2 v \rangle$. With such an approach, intersection types become little different than products, suggesting there is likely a problem. The erasure view immediately exposes this problem: with this evaluation rule a coercion can change the erasure of a term, in this case by introducing a new pair. Thus, this interpretation violates the spirit of the enterprise. A similar issue can be seen to arise with the `top` coercion; if `top` v evaluates to, say, `()`, then again the erasure changes.

Instead, the semantics rules that $\langle c_1, c_2 \rangle v$ is a value form, and any further "computation" with the coercions c_1 or c_2 is suspended until a projection coercion is applied. When projection occurs, $\pi_i(\langle c_1, c_2 \rangle v)$ evaluates to $c_i v$, and at no point does the erasure change. The value form $\langle c_1, c_2 \rangle v$ is most profitably read as a single value with two different views.[4]

[3]This view can be reconciled with languages in which types can be run-time objects by explicitly reflecting types into the term structure as in Crary, *et al.* [4].

[4]Dimock *et al.* [6] employ a similar idea: They include an intersection pair construct, and solve the above problem by requiring the erasure of e_1 and e_2 to be identical for $\langle e_1, e_2 \rangle$ to be *syntactically* well-formed. They can then define the erasure of an intersection pair

$$\begin{array}{lll} \textit{canonical} & & \\ \textit{values} \quad V & ::= & i \mid \lambda x{:}\tau.e \mid (v_1, v_2) \mid \Lambda\alpha.v \mid \\ & & \langle c_1, c_2\rangle v \mid \mathtt{top}[\tau]\, v \mid \mathtt{fold}[\mu\alpha.\tau]\, v \end{array}$$

$$\begin{array}{rll} \mathtt{id}\, v & \mapsto_c & v \\ (c_1 \circ c_2)v & \mapsto_c & c_1(c_2 v) \\ (c_1 \to c_2)(\lambda x{:}\tau.e) & \mapsto_c & \lambda x{:}\tau'.\, c_2(e[c_1 x/x]) \\ & & (\text{where } \vdash c_1 : \tau' \Rightarrow \tau) \\ (c_1 \times c_2)(v_1, v_2) & \mapsto_c & (c_1 v_1, c_2 v_2) \\ (\forall\alpha.c)(\Lambda\alpha.v) & \mapsto_c & \Lambda\alpha.\, c\, v \\ (\pi_i[\tau])(\langle c_1, c_2\rangle v) & \mapsto_c & c_i v \\ (\mathtt{app}[\tau']\,\tau)(\Lambda\alpha.v) & \mapsto_c & v[\tau/\alpha] \\ \mathtt{gen}\, v & \mapsto_c & \Lambda\alpha.v \quad (\alpha \text{ fresh}) \\ \mathtt{unfold}[\tau](\mathtt{fold}[\tau']v) & \mapsto_c & v \end{array}$$

$$\frac{v \mapsto_c v'}{c\, v \mapsto_c c\, v'} \; (c \text{ not } \langle -, -\rangle,\ \mathtt{top} \text{ or } \mathtt{fold})$$

(Rules for `rec` and `isorec` appear in Appendix B.2.)

Figure 13: Canonicalization

The semantics similarly defines $\mathtt{top}\, v$ and $\mathtt{fold}\, v$ to be value forms, and given this it proves to be convenient to say that $c\, v$ is a value for any coercion c. One pleasant consequence of this is that a term is a value exactly when its erasure is a value.

Another consequence of this design is that values do not enjoy useful canonical forms. For example, a functional value may have the form $\lambda x{:}\tau.e$, as one would prefer, but it may also have the form $\pi_1(\langle \mathtt{id}, c\rangle(\lambda x{:}\tau.e))$ or $(c_1 \to c_2)(\lambda x{:}\tau.e)$, for example. Therefore, the operational semantics utilizes two relations, the usual small-step evaluation relation (written $e \mapsto e'$), and an auxiliary, canonicalization relation (written $v \mapsto_c v'$) that converts values to canonical form. The canonicalization relation is purely a technical device; it represents no run-time action whatsoever, as formalized in Proposition 5.4 and Theorem 5.5.

For example, the rules for evaluating applications are as follows:

$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2} \qquad \frac{e \mapsto e'}{v\, e \mapsto v\, e'}$$

$$\frac{v_1 \mapsto_c v_1'}{v_1 v_2 \mapsto v_1' v_2} \qquad \frac{}{(\lambda x{:}\tau.e)v \mapsto e[v/x]}$$

The first, second, and fourth rules are standard. The third uses the canonicalization relation to place the function into canonical form so that the fourth rule can apply.

The canonical value forms and canonicalization rules are given in Figure 13. The remaining evaluation rules appear in Appendix B.2.

As usual, type safety follows from subject reduction and progress lemmas for evaluation, each of which requires a similar, auxiliary lemma for canonicalization:

Lemma 5.1 (Subject Reduction)

- *If* $\vdash v : \tau$ *and* $v \mapsto_c v'$ *then* $\vdash v' : \tau$.
- *If* $\vdash e : \tau$ *and* $e \mapsto e'$ *then* $\vdash e' : \tau$.

to be the common erasure of its components.

$$\begin{array}{rcl@{\qquad}rcl} x^\circ & \overset{\text{def}}{=} & x & (e.i)^\circ & \overset{\text{def}}{=} & (e^\circ).i \\ i^\circ & \overset{\text{def}}{=} & i & (\Lambda\alpha.v)^\circ & \overset{\text{def}}{=} & v^\circ \\ (\lambda x{:}\tau.e)^\circ & \overset{\text{def}}{=} & \lambda x.e^\circ & (c\, e)^\circ & \overset{\text{def}}{=} & e^\circ \\ (e_1 e_2)^\circ & \overset{\text{def}}{=} & e_1^\circ e_2^\circ & & & \end{array}$$

Figure 14: Erasure

Lemma 5.2 (Progress)

- *If* $\vdash v : \tau$ *then either* v *is canonical or* $v \mapsto_c v'$.
- *If* $\vdash e : \tau$ *then either* e *is a value or* $e \mapsto e'$.

Theorem 5.3 (Type Safety) *If* $\vdash e : \tau$ *and* $e \mapsto^* e'$ *then* e' *is not stuck (that is, either* e' *is a value or* $e' \mapsto e''$*).*

Two additional facts formalize the assertion that coercions have no run-time effect: First, as discussed above, canonicalization (*i.e.,* application of coercions) never affects a value's erasure. Second, any canonicalization sequence terminates in finitely many steps; this is important since nontermination is certainly a run-time effect. More importantly, without canonicalization the type safety result does not apply to the erased language, as nontermination of canonicalization could shield the typed language from an unsafe state that the erased language was able to reach.

Proposition 5.4 (Invariant Erasure) *Let* $(-)^\circ$ *be defined as in Figure 14. Then if* $v_1 \mapsto_c v_2$ *then* $v_1^\circ = v_2^\circ$.

Theorem 5.5 (Canonicalization) *If* $\vdash v : \tau$ *then* v *canonicalizes in a finite number of steps.*

Theorem 5.5 is proven using a logical relation argument that is detailed in Appendix D. It is worthwhile to note that this theorem depends on the typing condition; canonicalization can fail for ill-typed values. For example, $(\mathtt{unfold} \circ \mathtt{rec}(\chi.\, \mathtt{unfold} \circ \chi \circ \mathtt{fold}) \circ \mathtt{fold})v$ loops as it tries to canonicalize.

6 Conclusion

This work sheds new light on the Mitchell-Pierce interpretation of subtyping by showing that, in a type theory supporting implicit formation and elimination of quantified types, it is an entirely satisfactory encoding of full $F_\leq$ bounded quantification. By itself, this is a result of primarily theoretical importance (particularly given the intractable richness of the necessary type theory); however, with the calculus of explicit coercions this result becomes of practical interest to typed compilation.

By reifying subtyping derivations as explicit coercions, we not only provide an easily typechecked target language, but we also achieve this work's ultimate goal of compiling away subtyping entirely. Later phases of a typed compiler need deal only with the coercions, and some typed compilers (such as the Typed Assembly Language compiler [7]) support similar, if not quite so expressive, coercion constructs already. Moreover, by translating bounded quantification using intersection types rather than coercion abstractions, we allow the use of a relatively simple calculus of coercions.

A The Source Language

$\boxed{\Gamma \vdash_S \tau \text{ type}}$

$$\frac{}{\Gamma \vdash_S \tau \text{ type}} \; (FV(\tau) \subseteq \text{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash_S e : \tau}$

$$\frac{}{\Gamma \vdash_S x : \tau} \; ((x{:}\tau) \in \Gamma) \qquad \frac{}{\Gamma \vdash_S i : \texttt{int}}$$

$$\frac{\Gamma \vdash_S \tau_1 \text{ type} \quad \Gamma, x{:}\tau_1 \vdash_S e : \tau_2}{\Gamma \vdash_S \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \; (x \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash_S e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_S e_2 : \tau_1}{\Gamma \vdash_S e_1 e_2 : \tau_2}$$

$$\frac{\Gamma \vdash_S e_1 : \tau_1 \quad \Gamma \vdash_S e_2 : \tau_2}{\Gamma \vdash_S (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash_S e : \tau_1 \times \tau_2}{\Gamma \vdash_S e.i : \tau_i} \; (i = 1, 2)$$

$$\frac{\Gamma, \alpha{\leq}\tau \vdash_S v : \tau' \quad \Gamma \vdash_S \tau \text{ type}}{\Gamma \vdash_S \Lambda\alpha{\leq}\tau.v : \forall\alpha{\leq}\tau.\tau'} \; (\alpha \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash_S e : \forall\alpha{\leq}\tau_1.\tau_2 \quad \Gamma \vdash_S \tau \leq \tau_1}{\Gamma \vdash_S e[\tau] : \tau_2[\tau/\alpha]}$$

$$\frac{\Gamma \vdash_S e : \tau_1 \quad \Gamma \vdash_S \tau_1 \leq \tau_2}{\Gamma \vdash_S e : \tau_2}$$

$\boxed{\Gamma \vdash_S \tau_1 \leq \tau_2}$

$$\frac{\Gamma \vdash_S \tau \text{ type}}{\Gamma \vdash_S \tau \leq \tau}$$

$$\frac{\Gamma \vdash_S \tau_1 \leq \tau_2 \quad \Gamma \vdash_S \tau_2 \leq \tau_3}{\Gamma \vdash_S \tau_1 \leq \tau_3}$$

$$\frac{}{\Gamma \vdash_S \alpha \leq \tau} \; ((\alpha{\leq}\tau) \in \Gamma)$$

$$\frac{\Gamma \vdash_S \tau_1' \leq \tau_1 \quad \Gamma \vdash_S \tau_2 \leq \tau_2'}{\Gamma \vdash_S \tau_1 \to \tau_2 \leq \tau_1' \leq \tau_2'}$$

$$\frac{\Gamma \vdash_S \tau_1 \leq \tau_1' \quad \Gamma \vdash_S \tau_2 \leq \tau_2'}{\Gamma \vdash_S \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

$$\frac{\Gamma \vdash_S \tau_1' \leq \tau_1 \quad \Gamma, \alpha{\leq}\tau_1' \vdash_S \tau_2 \leq \tau_2'}{\Gamma \vdash_S \forall\alpha{\leq}\tau_1.\tau_2 \leq \forall\alpha{\leq}\tau_1'.\tau_2'}$$

$$\frac{\Gamma \vdash_S \tau \text{ type}}{\Gamma \vdash_S \tau \leq \texttt{top}}$$

$\boxed{\vdash_S \Gamma \text{ ok}}$

$$\frac{}{\vdash_S \epsilon \text{ ok}}$$

$$\frac{\vdash_S \Gamma \text{ ok} \quad \Gamma \vdash_S \tau \text{ type}}{\vdash_S \Gamma, \alpha{\leq}\tau \text{ ok}} \; (\alpha \notin \text{Dom}(\Gamma))$$

$$\frac{\vdash_S \Gamma \text{ ok} \quad \Gamma \vdash_S \tau \text{ type}}{\vdash_S \Gamma, x{:}\tau \text{ ok}} \; (x \notin \text{Dom}(\Gamma))$$

A.1 The First Target Language

To obtain the (first) target language from the source language, delete the variable subtyping rule, replace the rules for introduction, elimination, and subtyping of quantified types and for variable context formation by

$$\frac{\Gamma, \alpha \vdash_T v : \tau}{\Gamma \vdash_T \Lambda\alpha.v : \forall\alpha.\tau} \; (\alpha \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash_T e : \forall\alpha.\tau \quad \Gamma \vdash_T \tau' \text{ type}}{\Gamma \vdash_T e[\tau'] : \tau[\tau'/\alpha]}$$

$$\frac{\Gamma, \alpha \vdash_T \tau_1 \leq \tau_2}{\Gamma \vdash_T \forall\alpha.\tau_1 \leq \forall\alpha.\tau_2} \; (\alpha \notin \text{Dom}(\Gamma))$$

$$\frac{\vdash_T \Gamma \text{ ok}}{\vdash_T \Gamma, \alpha \text{ ok}} \; (\alpha \notin \text{Dom}(\Gamma))$$

and add the following rules:

$$\frac{\Gamma \vdash_T \tau_1 \wedge \tau_2 \text{ type}}{\Gamma \vdash_T \tau_1 \wedge \tau_2 \leq \tau_i} \; (i = 1, 2)$$

$$\frac{\Gamma \vdash_T \tau \leq \tau_1 \quad \Gamma \vdash_T \tau \leq \tau_2}{\Gamma \vdash_T \tau \leq \tau_1 \wedge \tau_2}$$

$$\frac{\Gamma \vdash_T \forall\alpha.\tau \text{ type} \quad \Gamma \vdash_T \tau' \text{ type}}{\Gamma \vdash_T \forall\alpha.\tau \leq \tau[\tau'/\alpha]}$$

$$\frac{\Gamma \vdash_T \tau \text{ type}}{\Gamma \vdash_T \tau \leq \forall\alpha.\tau} \; (\alpha \text{ not free in } \tau)$$

B The Coercion Calculus

B.1 Static Semantics

$\boxed{\Gamma \vdash \tau \text{ type}}$

$$\frac{}{\Gamma \vdash \tau \text{ type}} \; (FV(\tau) \subseteq \text{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash x : \tau} \; ((x{:}\tau) \in \Gamma) \qquad \frac{}{\Gamma \vdash i : \texttt{int}}$$

$$\frac{\Gamma \vdash \tau_1 \text{ type} \quad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \; (x \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.i : \tau_i} \; (i = 1, 2)$$

$$\frac{\Gamma, \alpha \vdash v : \tau}{\Gamma \vdash \Lambda\alpha.v : \forall\alpha.\tau} \; (\alpha \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma; \epsilon \vdash c : \tau_1 \Rightarrow \tau_2}{\Gamma \vdash c\, e : \tau_2}$$

$\boxed{\Gamma \vdash c : \tau_1 \Rightarrow \tau_2}$

$$\frac{\Gamma \vdash \tau \ \text{type}}{\Gamma;\Phi \vdash \texttt{id} : \tau \Rightarrow \tau}$$

$$\frac{\Gamma;\Phi \vdash c_1 : \tau_2 \Rightarrow \tau_3 \quad \Gamma;\Phi \vdash c_2 : \tau_1 \Rightarrow \tau_2}{\Gamma;\Phi \vdash c_1 \circ c_2 : \tau_1 \Rightarrow \tau_3}$$

$$\frac{\Gamma;\Phi \vdash c_1 : \tau_1' \Rightarrow \tau_1 \quad \Gamma;\Phi \vdash c_2 : \tau_2 \Rightarrow \tau_2'}{\Gamma;\Phi \vdash c_1 \to c_2 : (\tau_1 \to \tau_2) \Rightarrow (\tau_1' \to \tau_2')}$$

$$\frac{\Gamma;\Phi \vdash c_1 : \tau_1 \Rightarrow \tau_1' \quad \Gamma;\Phi \vdash c_2 : \tau_2 \Rightarrow \tau_2'}{\Gamma;\Phi \vdash c_1 \times c_2 : (\tau_1 \times \tau_2) \Rightarrow (\tau_1' \times \tau_2')}$$

$$\frac{(\Gamma,\alpha);\Phi \vdash c : \tau_1 \Rightarrow \tau_2}{\Gamma;\Phi \vdash \forall\alpha.c : \forall\alpha.\tau_1 \Rightarrow \forall\alpha.\tau_2} \ (\alpha \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma;\Phi \vdash c_1 : \tau \Rightarrow \tau_1 \quad \Gamma;\Phi \vdash c_2 : \tau \Rightarrow \tau_2}{\Gamma;\Phi \vdash \langle c_1, c_2\rangle : \tau \Rightarrow \tau_1 \wedge \tau_2}$$

$$\frac{\Gamma \vdash \tau_1 \wedge \tau_2 \ \text{type}}{\Gamma;\Phi \vdash \pi_i[\tau_1 \wedge \tau_2] : \tau_1 \wedge \tau_2 \Rightarrow \tau_i} \ (i = 1,2)$$

$$\frac{\Gamma \vdash \tau \ \text{type}}{\Gamma;\Phi \vdash \texttt{top}[\tau] : \tau \Rightarrow \texttt{top}}$$

$$\frac{\Gamma \vdash \forall\alpha.\tau \ \text{type} \quad \Gamma \vdash \tau' \ \text{type}}{\Gamma;\Phi \vdash \texttt{app}[\forall\alpha.\tau]\,\tau' : \forall\alpha.\tau \Rightarrow \tau[\tau'/\alpha]}$$

$$\frac{\Gamma \vdash \tau \ \text{type}}{\Gamma;\Phi \vdash \texttt{gen} : \tau \Rightarrow \forall\alpha.\tau} \ (\alpha \notin FV(\tau))$$

$$\frac{}{\Gamma;\Phi \vdash \chi : \tau_1 \Rightarrow \tau_2} \ ((\chi : \tau_1 \Rightarrow \tau_2) \in \Phi)$$

$$\frac{\begin{array}{c}\Gamma,\alpha \vdash \tau \ \text{type} \qquad \Gamma,\alpha' \vdash \tau' \ \text{type} \\ (\Gamma,\alpha,\alpha');(\Phi,\chi : \alpha \Rightarrow \alpha') \vdash c : \tau \Rightarrow \tau'\end{array}}{\Gamma;\Phi \vdash \texttt{rec}(\chi : \alpha \Rightarrow \alpha'.c) : \mu\alpha.\tau \Rightarrow \mu\alpha'.\tau'}$$
$$(\alpha, \alpha' \notin \text{Dom}(\Gamma), \chi \notin \text{Dom}(\Phi))$$

$$\frac{\begin{array}{cc}\Gamma,\alpha_1,\alpha_1' \vdash \tau_1^+ \ \text{type} & \Gamma,\alpha_1,\alpha_1' \vdash \tau_1^- \ \text{type} \\ \Gamma,\alpha_2,\alpha_2' \vdash \tau_2^+ \ \text{type} & \Gamma,\alpha_2,\alpha_2' \vdash \tau_2^- \ \text{type} \\ \Gamma';\Phi' \vdash c_+ : \tau_1^+ \Rightarrow \tau_2^+ & \Gamma';\Phi' \vdash c_- : \tau_2^- \Rightarrow \tau_1^-\end{array}}{\begin{array}{r}\Gamma;\Phi \vdash \texttt{isorec}(\chi_+ : \alpha_1 \Rightarrow \alpha_2', \chi_- : \alpha_2 \Rightarrow \alpha_1' . c_+, c_-) : \\ \mu\alpha_1.\tau_1 \Rightarrow \mu\alpha_2.\tau_2\end{array}}$$
$$\left(\begin{array}{l}\Gamma' = \Gamma, \alpha_1, \alpha_1', \alpha_2, \alpha_2' \\ \Phi' = \Phi, \chi_+ : \alpha_1 \Rightarrow \alpha_2', \chi_- : \alpha_2 \Rightarrow \alpha_1' \\ \alpha_1, \alpha_1', \alpha_2, \alpha_2' \notin \text{Dom}(\Gamma), \chi_+, \chi_- \notin \text{Dom}(\Phi) \\ \alpha_1 \text{ positive, } \alpha_1' \text{ negative in } \tau_1^+ \\ \alpha_2 \text{ negative, } \alpha_2' \text{ positive in } \tau_2^+ \\ \alpha_1 \text{ negative, } \alpha_1' \text{ positive in } \tau_1^- \\ \alpha_2 \text{ positive, } \alpha_2' \text{ negative in } \tau_2^- \\ \tau_1 = \tau_1^+[\alpha_1/\alpha_1'] = \tau_1^-[\alpha_1/\alpha_1'] \\ \tau_2 = \tau_2^+[\alpha_2/\alpha_2'] = \tau_2^-[\alpha_2/\alpha_2']\end{array}\right)$$

$$\frac{\Gamma;\Phi \vdash \mu\alpha.\tau \ \text{type}}{\Gamma;\Phi \vdash \texttt{fold}[\mu\alpha.\tau] : \tau[\mu\alpha.\tau/\alpha] \Rightarrow \mu\alpha.\tau}$$

$$\frac{\Gamma;\Phi \vdash \mu\alpha.\tau \ \text{type}}{\Gamma;\Phi \vdash \texttt{unfold}[\mu\alpha.\tau] : \mu\alpha.\tau \Rightarrow \tau[\mu\alpha.\tau/\alpha]}$$

$\boxed{\vdash \Gamma \ \text{ok}}$

$$\frac{}{\vdash \epsilon \ \text{ok}}$$

$$\frac{\vdash \Gamma \ \text{ok}}{\vdash \Gamma, \alpha \ \text{ok}} \ (\alpha \notin \text{Dom}(\Gamma))$$

$$\frac{\vdash \Gamma \ \text{ok} \quad \Gamma \vdash \tau \ \text{type}}{\vdash \Gamma, x{:}\tau \ \text{ok}} \ (x \notin \text{Dom}(\Gamma))$$

B.2 Dynamic Semantics

Evaluation

$$\frac{e \mapsto e'}{c\,e \mapsto c\,e'} \qquad \frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2} \qquad \frac{e \mapsto e'}{v\,e \mapsto v\,e'}$$

$$\frac{v_1 \mapsto_c v_1'}{v_1 v_2 \mapsto v_1' v_2} \qquad \frac{}{(\lambda x{:}\tau.e)v \mapsto e[v/x]}$$

$$\frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)} \qquad \frac{e \mapsto e'}{(v, e) \mapsto (v, e')}$$

$$\frac{e \mapsto e'}{e.i \mapsto e'.i} \qquad \frac{v \mapsto_c v'}{v.i \mapsto v'.i} \qquad \frac{}{(v_1, v_2).i \mapsto v_i} \ (i = 1,2)$$

Canonicalization

$$\frac{v \mapsto_c v'}{c\,v \mapsto_c c\,v'} \ (c \text{ not } \langle -, - \rangle, \ \texttt{top} \text{ or } \texttt{fold})$$

$$\begin{array}{rcl}
\texttt{id}\,v & \mapsto_c & v \\
(c_1 \circ c_2)v & \mapsto_c & c_1(c_2 v) \\
(c_1 \to c_2)(\lambda x{:}\tau.e) & \mapsto_c & \lambda x{:}\tau'.\, c_2(e[c_1 x/x]) \\
 & & (\text{where} \vdash c_1 : \tau' \Rightarrow \tau) \\
(c_1 \times c_2)(v_1, v_2) & \mapsto_c & (c_1 v_1, c_2 v_2) \\
(\forall\alpha.c)(\Lambda\alpha.v) & \mapsto_c & \Lambda\alpha.\, c\,v \\
(\pi_i[\tau])(\langle c_1, c_2\rangle v) & \mapsto_c & c_i v \\
(\texttt{app}[\tau']\,\tau)(\Lambda\alpha.v) & \mapsto_c & v[\tau/\alpha] \\
\texttt{gen}\,v & \mapsto_c & \Lambda\alpha.v \quad (\alpha \text{ fresh}) \\
\texttt{unfold}[\tau](\texttt{fold}[\tau']v) & \mapsto_c & v
\end{array}$$

$$\begin{array}{l}
\texttt{rec}(\chi : \alpha \Rightarrow \alpha'.c)(\texttt{fold}[\mu\alpha.\tau]v) \mapsto_c \\
\quad \texttt{fold}[\mu\alpha'.\tau'] \\
\quad ((c[\mu\alpha.\tau, \mu\alpha'.\tau', \texttt{rec}(\chi : \alpha \Rightarrow \alpha'.c)/\alpha, \alpha', \chi])\,v) \\
(\text{where} \vdash \texttt{rec}(\chi : \alpha \Rightarrow \alpha'.c) : \mu\alpha.\tau \Rightarrow \mu\alpha'.\tau')
\end{array}$$

$$\begin{array}{l}
\texttt{isorec}(\chi_+ : \alpha_1 \Rightarrow \alpha_2', \chi_- : \alpha_2 \Rightarrow \alpha_1' . c_+, c_-) \\
(\texttt{fold}[\mu\alpha_1.\tau_1]v) \\
\mapsto_c \\
\texttt{fold}[\mu\alpha_2.\tau_2] \\
(c_+[\mu\alpha_1.\tau_1, \mu\alpha_1.\tau_1, \mu\alpha_2.\tau_2, \mu\alpha_2.\tau_2, \\
\quad \texttt{isorec}(\chi_+ : \alpha_1 \Rightarrow \alpha_2', \chi_- : \alpha_2 \Rightarrow \alpha_1' . c_+, c_-), \\
\quad \texttt{isorec}(\chi_- : \alpha_2 \Rightarrow \alpha_1', \chi_+ : \alpha_1 \Rightarrow \alpha_2' . c_-, c_+)/ \\
\quad \alpha_1, \alpha_1', \alpha_2, \alpha_2', \chi_+, \chi_-]\,v) \\
(\text{where } \vdash \texttt{isorec}(\chi_+ : \alpha_1 \Rightarrow \alpha_2', \chi_- : \alpha_2 \Rightarrow \alpha_1' . c_+, c_-) : \\
\hfill \mu\alpha_1.\tau_1 \Rightarrow \mu\alpha_2.\tau_2)
\end{array}$$

C The Coercion Translation

$\boxed{\Gamma \vdash_{SC} e : \tau \ \rightrightarrows \ e'}$

$$\frac{}{\Gamma \vdash_{SC} x : \tau \ \rightrightarrows \ x} \ ((x : \tau) \in \Gamma)$$

$$\frac{}{\Gamma \vdash_{SC} i : \texttt{int} \ \rightrightarrows \ i}$$

$$\frac{\Gamma \vdash_S \tau_1 \ \text{type} \quad \Gamma, x{:}\tau_1 \vdash_{SC} e : \tau_2 \ \rightrightarrows \ e'}{\Gamma \vdash_{SC} \lambda x{:}\tau_1.e : \tau_1 \to \tau_2 \ \rightrightarrows \ \lambda x{:}|\tau_1|_\Gamma.e'} \ (x \notin \Gamma)$$

$$\frac{\Gamma \vdash_{SC} e_1 : \tau_1 \to \tau_2 \ \rightrightarrows \ e_1' \quad \Gamma \vdash_{SC} e_2 : \tau_1 \ \rightrightarrows \ e_2'}{\Gamma \vdash_{SC} e_1 e_2 : \tau_2 \ \rightrightarrows \ e_1' e_2'}$$

$$\frac{\Gamma \vdash_{SC} e_1 : \tau_1 \ \rightrightarrows \ e_1' \quad \Gamma \vdash_{SC} e_2 : \tau_2 \ \rightrightarrows \ e_2'}{\Gamma \vdash_{SC} (e_1, e_2) : \tau_1 \times \tau_2 \ \rightrightarrows \ (e_1', e_2')}$$

$$\frac{\Gamma \vdash_{SC} e : \tau_1 \times \tau_2 \ \rightrightarrows\ e'}{\Gamma \vdash_{SC} e.i : \tau_i \ \rightrightarrows\ e'.i} \ (i = 1, 2)$$

$$\frac{\Gamma, \alpha{\leq}\tau \vdash_{SC} v : \tau' \ \rightrightarrows\ v' \quad \Gamma \vdash_S \tau \text{ type}}{\Gamma \vdash_{SC} v : \forall\alpha{\leq}\tau.\tau' \ \rightrightarrows\ \Lambda\alpha.v'} \ (\alpha \notin \Gamma)$$

$$\frac{\Gamma \vdash_{SC} e : \forall\alpha{\leq}\tau_1.\tau_2 \ \rightrightarrows\ e' \quad \Gamma \vdash_{SC} \tau \leq \tau_1 \ \rightrightarrows\ c}{\begin{array}{l}\Gamma \vdash_{SC} e[\tau] : \tau_2[\tau/\alpha] \ \rightrightarrows \\ \qquad (\mathtt{map}[\alpha.|\tau_2|_\Gamma](\pi_1, \langle \mathtt{id}, c\rangle) \circ \mathtt{app}\,|\tau|_\Gamma)\, e'\end{array}} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash_{SC} e : \tau_1 \ \rightrightarrows\ e \quad \Gamma \vdash_{SC} \tau_1 \leq \tau_2 \ \rightrightarrows\ c}{\Gamma \vdash_{SC} e : \tau_2 \ \rightrightarrows\ c\, e}$$

$$\boxed{\Gamma \vdash_{SC} \tau_1 \leq \tau_2 \ \rightrightarrows\ c}$$

$$\frac{\Gamma \vdash_S \tau \text{ type}}{\Gamma \vdash_{SC} \tau \leq \tau \ \rightrightarrows\ \mathtt{id}}$$

$$\frac{\Gamma \vdash_{SC} \tau_1 \leq \tau_2 \ \rightrightarrows\ c_1 \quad \Gamma \vdash_{SC} \tau_2 \leq \tau_3 \ \rightrightarrows\ c_2}{\Gamma \vdash_{SC} \tau_1 \leq \tau_3 \ \rightrightarrows\ c_2 \circ c_1}$$

$$\frac{}{\Gamma \vdash_{SC} \alpha \leq \tau \ \rightrightarrows\ \pi_2} \ ((\alpha{\leq}\tau) \in \Gamma)$$

$$\frac{\Gamma \vdash_{SC} \tau_1' \leq \tau_1 \ \rightrightarrows\ c_1 \quad \Gamma \vdash_{SC} \tau_2 \leq \tau_2' \ \rightrightarrows\ c_2}{\Gamma \vdash_{SC} (\tau_1 \to \tau_2) \leq (\tau_1' \to \tau_2') \ \rightrightarrows\ (c_1 \to c_2)}$$

$$\frac{\Gamma \vdash_{SC} \tau_1 \leq \tau_1' \ \rightrightarrows\ c_1 \quad \Gamma \vdash_{SC} \tau_2 \leq \tau_2' \ \rightrightarrows\ c_2}{\Gamma \vdash_{SC} \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2' \ \rightrightarrows\ c_1 \times c_2}$$

$$\frac{\Gamma \vdash_{SC} \tau_1' \leq \tau_1 \ \rightrightarrows\ c_1 \quad \Gamma, \alpha{\leq}\tau_1' \vdash_{SC} \tau_2 \leq \tau_2' \ \rightrightarrows\ c_2}{\begin{array}{l}\Gamma \vdash_{SC} \forall\alpha{\leq}\tau_1.\tau_2 \leq \forall\alpha{\leq}\tau_1'.\tau_2' \ \rightrightarrows \\ \qquad \forall\alpha.(c_2 \circ \\ \qquad\qquad \mathtt{map}[\alpha.|\tau_2|_\Gamma](\pi_1, \langle \mathtt{id}, c_1 \circ \pi_2\rangle) \circ \\ \qquad\qquad \mathtt{app}\,(\alpha \wedge |\tau_1'|_\Gamma)) \circ \\ \qquad \mathtt{gen}\end{array}} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash_S \tau \text{ type}}{\Gamma \vdash_{SC} \tau \leq \mathtt{top} \ \rightrightarrows\ \mathtt{top}}$$

D Canonicalization Proof

We define a set of closed values S to be *canonically closed* when:

- if $v \in S$ and $v \mapsto_c v'$ then $v' \in S$, and
- if $v \in S$ and $v' \mapsto_c v$ and v' has the same type as v then $v' \in S$.

We define an *assignment* to be a finite mapping from type variables to pairs (τ, S), such that τ is a closed type, and S is a canonically closed set of closed values having type τ. If σ is an assignment, we write $\sigma(v)$ to mean the result of performing the substitution obtained by ignoring the set components of the assignment.

We next define a logical relation R^σ_τ over values to be the least relation such that $R^\sigma_\tau(v)$ holds if $\vdash v : \sigma(\tau)$, v canonicalizes, and:

- if τ is α then $v \in S$ where $\sigma(\alpha) = (\tau, S)$
- if τ is $\tau_1 \wedge \tau_2$ then $R^\sigma_{\tau_1}(\pi_1 v)$ and $R^\sigma_{\tau_2}(\pi_2 v)$
- if τ is $\forall\alpha.\tau'$ then for any closed type τ'', $R^\sigma_{\tau'[\tau''/\alpha]}((\mathtt{app}\,\tau'')v)$
- if τ is $\mu\alpha.\tau'$ then $R^\sigma_{\tau'[\tau/\alpha]}(\mathtt{unfold}\,v)$.

Note that function spaces (and products as well) are considered to be base types, which dramatically simplifies the construction of the relation, as it eliminates any negative appearances of the relation being defined. This is possible because there exist no coercions that eliminate functions or products.

Lemma D.1 *When R^σ_τ is well-defined (that is, when the $FV(\tau) \subseteq \mathrm{Dom}(\sigma)$), it is canonically closed.*

Define the relation $R^\sigma_{\tau_1 \Rightarrow \tau_2}$ over coercions so that $R^\sigma_{\tau_1 \Rightarrow \tau_2}(c)$ iff $R^\sigma_{\tau_1}(v)$ implies $R^\sigma_{\tau_2}(c\,v)$, for all values v. Let $[\![\Gamma]\!]$ be the set of assignments whose domain is exactly the domain of Γ. Also let $[\![\Phi]\!]\sigma$ be the set of mappings φ from $\mathrm{Dom}(\Phi)$ to closed coercions such that $R^\sigma_{\Phi(\chi)}(\varphi(\chi))$, for all $\chi \in \mathrm{Dom}(\Phi)$.

Lemma D.2 *If $\Gamma; \Phi \vdash c : \tau_1 \Rightarrow \tau_2$ and $\sigma \in [\![\Gamma]\!]$ and $\varphi \in [\![\Phi]\!]\sigma$ then $R^\sigma_{\tau_1 \Rightarrow \tau_2}(\varphi(\sigma(c)))$.*

Proof

By induction on derivations. The interesting cases are those for `rec` and `isorec`.

Case 1: Suppose the last rule applied is `rec` formation, and let $\sigma \in [\![\Gamma]\!]$ and $\varphi \in [\![\Phi]\!]\sigma$. Suppose $R^\sigma_{\mu\alpha.\tau}(v)$. We wish to show that $R^\sigma_{\mu\alpha'.\tau'}(\varphi(\sigma(\mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c)))\,v)$. We show this by induction on the derivation of $R^\sigma_{\mu\alpha.\tau}(v)$.

Let $c' = \varphi(\sigma(c))$. By assumption, v canonicalizes and has appropriate type, so $v \mapsto^*_c \mathtt{fold}\,v'$. Thus $\mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c')\,v$ canonicalizes and it certainly has appropriate type.

It remains to show that $R^\sigma_{\tau'[\mu\alpha'.\tau'/\alpha]}(\mathtt{unfold}(\mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c')\,v))$. Let S be the set of values v' such that $R^\sigma_{\mu\alpha.\tau}(v')$ follows from a smaller derivation than that of $R^\sigma_{\mu\alpha.\tau}(v)$. It is not difficult to show that S is canonically closed. Let σ' extend σ so that $\sigma'(\alpha) = (\sigma(\mu\alpha.\tau), S)$ and $\sigma'(\alpha') = (\sigma(\mu\alpha'.\tau'), R^\sigma_{\mu\alpha'.\tau'})$. Also let φ' extend φ so that $\varphi'(\chi) = \mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c')$. Note that $\sigma' \in [\![\Gamma, \alpha, \alpha']\!]$.

By induction, $R^\sigma_{\mu\alpha'.\tau'}(\mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c')\,v')$ for all $v' \in S$, and thus $R^{\sigma'}_{\alpha \Rightarrow \alpha'}(\mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c'))$. Thus $\varphi' \in [\![\Psi, \chi : \alpha \Rightarrow \alpha']\!]\sigma'$. Hence, by the outer induction hypothesis, $R^{\sigma'}_{\tau \Rightarrow \tau'}(\varphi'(\sigma'(c)))$. Let $c'' = \varphi'(\sigma'(c)) = c'[\sigma(\mu\alpha.\tau), \sigma(\mu\alpha'.\tau'), \mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c')/\alpha, \alpha', \chi]$.

We may show by induction on τ that $R^{\sigma'}_\tau(\mathtt{unfold}\,v)$ (since $R^\sigma_{\tau[\mu\alpha.\tau/\alpha]}(\mathtt{unfold}\,v)$ by a smaller derivation than $R^\sigma_{\mu\alpha.\tau}(v)$). Thus, using the above, $R^{\sigma'}_{\tau'}(c''(\mathtt{unfold}\,v))$. We may then show, by induction on τ, that $R^\sigma_{\tau'[\mu\alpha'.\tau'/\alpha']}(c''(\mathtt{unfold}\,v))$. Observe that:

$$\begin{array}{ll} & \mathtt{unfold}(\mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c')\,v) \\ \mapsto^*_c & \mathtt{unfold}(\mathtt{rec}(\chi : \alpha \Rightarrow \alpha'.c')\,(\mathtt{fold}\,v')) \\ \mapsto_c & \mathtt{unfold}(\mathtt{fold}(c''v')) \\ \mapsto_c & c''v' \end{array}$$

and

$$\begin{array}{rcl} c''(\texttt{unfold}\, v) & \mapsto_c^* & c''(\texttt{unfold}(\texttt{fold}\, v')) \\ & \mapsto_c & c'' v' \end{array}$$

Therefore, $R^{\sigma}_{\tau'[\mu\alpha'.\tau'/\alpha']}(\texttt{unfold}(\texttt{rec}(\chi : \alpha \Rightarrow \alpha'.c')\, v))$ by canonical closure.

Case 2: Suppose the last rule applied is `isorec` formation. The reasoning in this case is similar to the previous one, except that σ' and φ' are defined so that:

$$\begin{array}{rcl} \sigma'(\alpha_1) & = & (\sigma(\mu\alpha_1.\tau_1), S) \\ \sigma'(\alpha_1') & = & (\sigma(\mu\alpha_1.\tau_1), R^{\sigma}_{\mu\alpha_1.\tau_1}) \\ \sigma'(\alpha_2) & = & (\sigma(\mu\alpha_2.\tau_2), \emptyset) \\ \sigma'(\alpha_2') & = & (\sigma(\mu\alpha_2.\tau_2), R^{\sigma}_{\mu\alpha_2.\tau_2}) \\ \varphi'(\chi_+) & = & \texttt{isorec}(\chi_+ : \alpha_1 \Rightarrow \alpha_2', \chi_- : \alpha_2 \Rightarrow \alpha_1'. \\ & & \qquad \varphi(\sigma(c_+)), \varphi(\sigma(c_-))) \\ \varphi'(\chi_-) & = & \texttt{isorec}(\chi_- : \alpha_2 \Rightarrow \alpha_1', \chi_+ : \alpha_1 \Rightarrow \alpha_2'. \\ & & \qquad \varphi(\sigma(c_-)), \varphi(\sigma(c_+))) \end{array}$$

where S is defined analogously to the previous case. Then $R^{\sigma'}_{\alpha_1 \Rightarrow \alpha_2'}(\varphi'(\chi_+))$ holds by the same sort of induction as in the previous case, and $R^{\sigma'}_{\alpha_2 \Rightarrow \alpha_1'}(\varphi'(\chi_-))$ holds vacuously.

Using Lemma D.2, we may easily prove Theorem 5.5 by an induction on typing derivations.

References

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

[2] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2), 1994.

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

[4] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998. Extended version published as Cornell University technical report TR98-1721.

[5] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

[6] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *1997 ACM International Conference on Functional Programming*, pages 11–24, Amsterdam, June 1997.

[7] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[8] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[9] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1991.

[10] Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, April 1997.