Positive Subtyping

Martin Hofmann* **Benjamin Pierce†**

To appear in Information and Computation

The statement $S \leq T$ in a λ-calculus with subtyping is traditionally interpreted by a semantic coercion function of type $[\![S]\!] \rightarrow [\![T]\!]$ that extracts the "T part" of an element of S. If the subtyping relation is restricted to covariant positions, this interpretation may be enriched to include both the implicit coercion and an overwriting function $put[S,T] \in [\![S]\!] \rightarrow [\![T]\!] \rightarrow [\![S]\!]$ that updates the T part of an element of S. We give a realizability model and a sound equational theory for a second-order calculus of positive subtyping.

Though weaker than familiar calculi of bounded quantification, positive subtyping retains sufficient power to model objects, encapsulation, and message passing, and inheritance. The equational laws relating the behavior of coercions and *put* functions can be used to prove simple properties of the resulting classes in such a way that proofs for superclasses are "inherited" by subclasses.

1 Introduction

The syntactic device of subtyping reifies a number of related semantic intuitions. If types are regarded simply as predicates — sets of values — then the statement $S \leq T$ asserts that $[\![S]\!] \subseteq [\![T]\!]$: the set denoted by S is a subset of the one denoted by T. From a more refined point of view, we can think of S as a "more informative" or "richer" type than T. In the standard example, S and T are record types where S has all of the fields of T and possibly others. Then viewing a value $s \in S$ as an element of T involves projecting out the "T part" of s. Indeed, we may want the coercion from $[\![S]\!]$ to $[\![T]\!]$ to involve literally throwing away the irrelevant parts of s, so that two elements of $[\![S]\!]$ that differ only on fields not present in T will be judged equal as elements of T.

It has been noticed that these simple interpretations of subtyping do not offer satisfactory support for programming with *update*. Consider a standard problem that arises when a λ-calculus with subtyping is used to model a purely functional fragment of Smalltalk. An *object* in such a model is a record of instance variables together with a collection of functions (its *methods*) that can be invoked to perform various transformations and inquiries on the instance variables. For example, the instance variables of a one-dimensional point could be represented by a one-field record of type $\{x{:}\mathit{Int}\}$, whereas a colored point object would use a richer representation type like $\{x{:}\mathit{Int},\ c{:}\mathit{Color}\}$. Suppose that both kinds of points have a *bump* method that increments the x field. In the case of ordinary points, this method would have the type $\{x{:}\mathit{Int}\} \rightarrow \{x{:}\mathit{Int}\}$; colored points would come with a *bump* method of type $\{x{:}\mathit{Int},\ c{:}\mathit{Color}\} \rightarrow \{x{:}\mathit{Int},\ c{:}\mathit{Color}\}$. Now, a characteristic feature

*Fachbereich Mathematik, Technische Hochschule Darmstadt, Schloßgartenstraße 7, D-64289 Darmstadt, Germany. Electronic mail: `mh@mathematik.th-darmstadt.de`

†University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, U.K. Electronic mail: `bcp@dcs.ed.ac.uk`.

of object-oriented programming languages is the ability to define the common behavior of points and colored points only once: we want to write a *class* of points, from which point objects may be instantiated, and use this class to build a *subclass* of colored points where just the behavior for colors is added. In particular, we want to write the *bump* method just once.

Since instances of the point and colored point classes have *bump* methods of different types, a single definition of *bump* must clearly be polymorphic in the type of the state, i.e. it should have a type like

$$bump \in All\,(X{\leq}\{x{:}Int\})\; X{\rightarrow}X,$$

as suggested by Cardelli and Wegner [1985]. Unfortunately, given the standard interpretation of subtyping, this type is not inhabited by any useful functions [Robinson and Tennent, 1988]. In effect, the constraint $X{\leq}\{x{:}Int\}$ is too weak to allow an element e of X to be manipulated in any nontrivial way, aside from throwing away all the information in e except the x field. In particular, there is no way to construct a new element of X with an updated x field. (More formally, in parametric models, where the behavior of polymorphic functions is independent of the type at which they are instantiated, types like $All\,(X{\leq}T)\; X{\rightarrow}X$ are inhabited only by identity functions. When an element f is instantiated at some one-element subset S of T, the resulting function from S to S must clearly be the identity. But all instances of f have the same behavior, so they are all identities.)

This deficiency has led to proposals for enriching the language of types so that in-place modifications of records may be given sound typings [Cardelli and Mitchell, 1991, Cardelli, 1992]. But the complexity of these extensions has hindered their widespread acceptance. We propose here a more radical approach: modify the semantics of subtyping to include both projection and update, so that the naive polymorphic typing of *bump* becomes sound.

Rather than a simple coercion function from $[\![S]\!]$ to $[\![T]\!]$, we interpret the statement $S \leq T$ by a *pair* of functions,

$$[\![S \leq T]\!] \quad \in \quad \underbrace{[\![S]\!] \rightarrow [\![T]\!]}_{\text{implicit coercion}} \;\times\; \underbrace{[\![S]\!] \rightarrow [\![T]\!] \rightarrow [\![S]\!]}_{\text{put[S,T]}},$$

one for projecting out the T part of an element of S and one for overwriting the T part of an existing element of S with a new element of T. As usual, we elide uses of the "forgetful coercion" in the concrete syntax of programs; the second is denoted by a constant $put[S,T]$. For example,

$$\begin{array}{ll} & put[\{x{:}Int,\ c{:}Color\}, \{x{:}Int\}] \\ & \qquad \{x{=}5,\ c{=}blue\} \\ & \qquad \{x{=}6\} \\ = & \{x{=}6,\ c{=}blue\}. \end{array}$$

The coercion and update functions are related by three laws:

1. Updating a value $s \in S$ by $t \in T$ and then projecting out the T part yields exactly t.
2. Updating s with the T part of s itself leaves s unchanged.
3. Updating s with $t_1 \in T$ and then with $t_2 \in T$ yields the same result as performing just the second update.

Laws similar to these arise in Oles' category of "state shapes" [1985]. This coincidence [pointed out to us by John Reynolds and Bob Tennent] is reassuring, since Oles is also concerned with the semantics of update, though he works in a setting that does not involve subtyping *per se*.

The simple intuition of overwriting records extends naturally to the results of functions and polymorphic functions. To overwrite one function by another, for example, we form a new function that applies both of the original functions to its argument and then overwrites one result with the other. However, this construction works only in *result* positions; in general, it does not make sense when subtyping is allowed in contravariant positions like the domains of functions or the bounds of quantifiers. This leads us to the idea of a *positive subtyping* calculus in which subtyping is allowed only in covariant positions and the refined interpretation of subtyping as coercion plus update always makes sense.

For purposes of this study, we focus on the pure form of this calculus, in which *only* positive subtyping is allowed. One can also imagine hybrid subtyping systems, in which both ordinary and positive subtyping coexist. We return to this point in Section 7.

Other recent papers show a related tendency to tune the definition of subtyping to achieve soundness of various forms of updating constructs. Abadi and Cardelli's calculus of primitive objects [1994], for example, allows individual methods of objects to be replaced in running programs. Soundness of this update is achieved by interpreting an object type by a semantic union over fixed points of all possible extensions of its methods; the corresponding subtyping rule for object types allows extension of the set of methods but no refinement of the types of existing methods. Bruce [1994] allows methods to update instance variables, again restricting subtyping to extensions of the collection of instance variables. In these systems, the entities subject to update are complex data structures, governed by specially tailored rules. We take a more elementary approach, first studying update in a general setting and later applying our treatment to add high-level update to (a second-order fragment of) the primitive object model proposed by Pierce and Turner [1994].

In Sections 2, 3, and 4, we introduce the syntax, equational theory, and semantics of a positive variant of the calculus $F_{\leq}$ of second-order bounded quantification [Cardelli *et al.*, 1994, Curien and Ghelli, 1992]. Section 5 gives a small example, showing how updateable records can be encoded in this system.

Section 6 presents an example at some length, showing how the equational laws of positive $F_{\leq}$ can be used to prove nontrivial facts about a small object-oriented program. This proof is *modular*, in the same sense that the object-oriented program itself is modular. The program contains two class definitions, the second inheriting some of its behavior from the first. Although both classes involve "recursive self-reference" through the pseudo-variable *self*, the proof can be structured so that properties of the second class can be established without looking back at the implementation of the first class.

Section 7 discusses some limitations imposed by the restriction to positive-only subtyping and briefly speculates on possible extensions. With only positive subtyping one loses "subsumption for object types" (while retaining the ability to write programs that operate polymorphically over objects with different signatures); however, it appears that positive and ordinary subtyping can peacefully coexist, in the sense that $S \leq T$ can be allowed even when its proof involves contravariance; the constant $put[S,T]$ is simply left undefined in these cases. Moreover, we foresee no barriers to integrating features such as higher-order subtyping [Cardelli, 1990, Compagnoni, 1994, Pierce and Steffen, 1995] and partial functions, extending our simple example to a full-scale model of object-oriented programming.

Although our development is self-contained, readers may like to compare our system with standard bounded quantification calculi [Curien and Ghelli, 1992, Cardelli *et al.*, 1994] and semantic models of subtyping [Bruce and Longo, 1990, Breazu-Tannen *et al.*, 1991]. In Section 6, familiarity with the literature on type-theoretic models of object-oriented programming languages [Fisher and Mitchell, 1994, Abadi and Cardelli, 1994, Bruce, 1994, Pierce and Turner, 1994, etc.] may be helpful. Readers whose primary interest is in the application of positive subtyping to objects and inheritance may want to skip some of the earlier technical development, skimming just the definitions in Sections 2, 3, and 5 and then reading carefully from Section 6.

2 Definitions

We begin with the concrete syntax and typing rules of $F_{\leq}$, extended with a cartesian product type and a family of constants $put[S,T]$ and restricted to positive subtyping. (We take products as primitive because, under the restriction to positive subtyping, their polymorphic encodings do not validate the desired subtyping laws; cf. Section 7.)

The sets of terms, types, and contexts are:

$$
\begin{array}{lll}
e & ::= & x \;\mid\; \mathit{fun}\,(x{:}T)\,e \;\mid\; e_1\; e_2 \;\mid\; \mathit{fun}\,(X{\leq}T)\,e \;\mid\; e\,T \;\mid \\
 & & \langle e_1,\, e_2\rangle \;\mid\; e.1 \;\mid\; e.2 \;\mid\; \mathit{put}[T_1, T_2] \\[1ex]
T & ::= & \mathit{Top} \;\mid\; X \;\mid\; T_1{\to}T_2 \;\mid\; \mathit{All}\,(X{\leq}T_1)\,T_2 \;\mid\; T_1{\times}T_2 \\[1ex]
\Gamma & ::= & \emptyset \;\mid\; \Gamma,\, X{\leq}T \;\mid\; \Gamma,\, x{:}T
\end{array}
$$

The judgements of the system are statements of the form $\Gamma \vdash S \leq T$ (subtyping), $\Gamma \vdash e \in T$ (typing), and $\Gamma \vdash e_1 = e_2 \in T$ (equality) in which the free variables on the right of the turnstile are all bound in Γ and the free type variables in each binding in Γ are bound to the left; we formally identify statements up to renaming of variables bound on either the right or the left of the turnstile. This is equivalent to regarding alphabetic variable names as informal abbreviations for an underlying representation based on de Bruijn indices [de Bruijn, 1972], and implies the usual conventions about name capture during substitution, alpha-conversion, side-conditions concerning freshness of names, etc. It also follows from this point of view that the names bound by a context Γ are always taken to be pairwise distinct, which justifies an abuse of notation whereby Γ is regarded as a finite function from term and type variables to types. The capture-avoiding substitution of e_1 for x in e_2 is written $[e_1/x]e_2$; substitution of types is written $[S/X]T$.

In examples, we also use records and base types like *Int* and *Color*. Base types and constants may be regarded as variables in some standard pervasive context. Records are discussed in Section 5 and existential types in Section 7.

2.1 Subtyping

The subtyping relation $\Gamma \vdash S \leq T$ (pronounced "S is a subtype of T under assumptions Γ") is the least relation closed under the following rules:

$$\Gamma \vdash S \leq \mathit{Top} \tag{S-Top}$$

$$\Gamma \vdash X \leq \Gamma(X) \tag{S-TVar}$$

$$\Gamma \vdash S \leq S \tag{S-Refl}$$

$$\frac{\Gamma \vdash S \leq R \qquad \Gamma \vdash R \leq T}{\Gamma \vdash S \leq T} \tag{S-Trans}$$

$$\frac{\Gamma \vdash S_2 \leq T_2}{\Gamma \vdash U \to S_2 \leq U \to T_2} \tag{S-Arrow}$$

$$\frac{\Gamma,\, X {\leq} U \vdash S_2 \leq T_2}{\Gamma \vdash \mathit{All}\,(X {\leq} U)\, S_2 \leq \mathit{All}\,(X {\leq} U)\, T_2} \tag{S-All}$$

$$\frac{\Gamma \vdash S_1 \leq T_1 \qquad \Gamma \vdash S_2 \leq T_2}{\Gamma \vdash S_1 {\times} S_2 \leq T_1 {\times} T_2} \tag{S-Prod}$$

Except for S-Arrow, these are exactly the subtyping rules of $F_{\leq}$ (more precisely, of the *Kernel Fun* variant of $F_{\leq}$ [Cardelli and Wegner, 1985]; full $F_{\leq}$ [Curien and Ghelli, 1992, Cardelli *et al.*, 1994] uses a richer, but problematic, version of S-All; see [Pierce, 1994, Pierce and Steffen, 1995]). S-Top asserts that *Top* is a maximal element of the subtype ordering for every Γ; S-TVar uses an assumption of the form $X{\leq}T$ from the context; S-Refl and S-Trans state that the subtype relation is a preorder; S-Arrow, S-All, and S-Prod extend the relation to functions, polymorphic functions, and cartesian products. Note that S-Arrow and S-All are both *non*-variant on the left-hand side and covariant on the right; S-Prod is covariant in both positions as usual.

We shall need a few proof-theoretic facts about this definition. Most importantly, it can be shown that, for every derivable subtyping judgement $\Gamma \vdash S \leq T$, there is a derivation with this conclusion in which the reflexivity rule is used only on variables and the transitivity rule is used only with the left-hand hypothesis being an axiom promoting a type variable to its upper bound.

2.1.1 Definition: A derivation d of a statement $\Gamma \vdash S \leq T$ is *algorithmic* if

1. $T = \mathit{Top}$ and d is an instance of S-Top; or
2. $S = X$ and d is either an instance of reflexivity or an instance of transitivity whose left-hand subderivation is an instance of S-TVar and whose right-hand subderivation is algorithmic; or
3. $S = U {\to} S_2$ and $T = U {\to} T_2$ and d is an instance of S-Arrow whose subderivation is algorithmic; or
4. $S = \mathit{All}\,(X {\leq} U)\, S_2$ and $T = \mathit{All}\,(X {\leq} U)\, T_2$ and d is an instance of S-All whose subderivation is algorithmic; or
5. $S = S_1 {\times} S_2$ and $T = T_1 {\times} T_2$ and d is an instance of S-Prod both of whose subderivations are algorithmic.

Note that this definition is proper because it proceeds by induction on derivations, which are finite structures. The term "algorithmic" is intended to suggest that these derivations correspond to the succeeding traces of a decision procedure for the subtype relation.

2.1.2 Fact: If $\Gamma \vdash S \leq T$, then there is an algorithmic derivation with this conclusion.

Proof: Easy simplification of the standard proof [Curien and Ghelli, 1992, Cardelli *et al.*, 1994]. □

One application of this fact will be needed in the next section:

2.1.3 Lemma: If $\Gamma \vdash \mathit{All}\,(X{\leq}S_1)\,S_2 \leq \mathit{All}\,(X{\leq}T_1)\,T_2$, then $S_1 = T_1$.

Proof: By 2.1.2, there is an algorithmic derivation of $\Gamma \vdash \mathit{All}\,(X{\leq}S_1)\,S_2 \leq \mathit{All}\,(X{\leq}T_1)\,T_2$. But by the definition of "algorithmic," this derivation must end with an instance of S-ALL, which can only be the case if $S_1 = T_1$. □

2.2 Typing

The typing relation $\Gamma \vdash e \in T$ is exactly the same as in standard $F_{\leq}$ except for the constants $put[S,T]$. It is the least relation closed under the following rules.

$$\Gamma \vdash x \in \Gamma(x) \tag{T-VAR}$$

$$\frac{\Gamma,\, x{:}S_1 \vdash e \in S_2}{\Gamma \vdash \mathit{fun}\,(x{:}S_1)\; e \in S_1{\rightarrow}S_2} \tag{T-ARROW-I}$$

$$\frac{\Gamma \vdash e_1 \in S_1{\rightarrow}S_2 \qquad \Gamma \vdash e_2 \in S_1}{\Gamma \vdash e_1\; e_2 \in S_2} \tag{T-ARROW-E}$$

$$\frac{\Gamma,\, X{\leq}S_1 \vdash e \in S_2}{\Gamma \vdash \mathit{fun}\,(X{\leq}S_1)\; e \in \mathit{All}\,(X{\leq}S_1)\,S_2} \tag{T-ALL-I}$$

$$\frac{\Gamma \vdash e \in \mathit{All}\,(X{\leq}S_1)\,S_2 \qquad \Gamma \vdash T \leq S_1}{\Gamma \vdash e\; T \in [T/X]S_2} \tag{T-ALL-E}$$

$$\frac{\Gamma \vdash e_1 \in S_1 \qquad \Gamma \vdash e_2 \in S_2}{\Gamma \vdash \langle e_1,\, e_2\rangle \in S_1{\times}S_2} \tag{T-PROD-I}$$

$$\frac{\Gamma \vdash e \in S_1{\times}S_2}{\Gamma \vdash e.1 \in S_1} \tag{T-PROD-E1}$$

$$\frac{\Gamma \vdash e \in S_1{\times}S_2}{\Gamma \vdash e.2 \in S_2} \tag{T-PROD-E2}$$

$$\frac{\Gamma \vdash S \leq T}{\Gamma \vdash put[S,T] \in S{\rightarrow}T{\rightarrow}S} \tag{T-PUT}$$

$$\frac{\Gamma \vdash e \in S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash e \in T} \tag{T-SUB}$$

The rule T-VAR uses a typing assumption from the context; T-ARROW-I, T-ALL-I, T-ARROW-E, and T-ALL-E are the standard rules for introduction and elimination of functional and quantified types; T-PROD-I, T-PROD-E1, and T-PROD-E2 give pairing and projection; T-PUT allows $put[S,T]$ to be used as a function of the appropriate shape whenever $S \leq T$; T-SUB is the rule of *subsumption* characteristic of λ-calculi with subtyping [Cardelli, 1984, Reynolds, 1985].

Again, we shall need a few simple facts about this definition.

2.2.1 Definition: Let d be a derivation of a statement $\Gamma \vdash e \in S$. Then

- d is *minimal* if $\Gamma \vdash e \in T$ implies $\Gamma \vdash S \leq T$;
- d is *arrow-minimal* if $\Gamma \vdash e \in T_1 \rightarrow T_2$ implies $\Gamma \vdash S \leq T_1 \rightarrow T_2$;
- d is *All-minimal* if $\Gamma \vdash e \in \mathit{All}\,(X{\leq}T_1)\,T_2$ implies $\Gamma \vdash S \leq \mathit{All}\,(X{\leq}T_1)\,T_2$;
- d is *product-minimal* if $\Gamma \vdash e \in T_1 \times T_2$ implies $\Gamma \vdash S \leq T_1 \times T_2$.

2.2.2 Definition: A derivation d of a statement $\Gamma \vdash e \in T$ is *algorithmic* if

- d's last rule is T-VAR or T-PUT; or
- d's last rule is T-ARROW-I, T-ALL-I, or T-PROD-I with all subderivations minimal; or
- d's last rule is T-ARROW-E with an arrow-minimal subderivation on the left and a minimal subderivation on the right; or
- d's last rule is T-ALL-E with an All-minimal subderivation on the left; or
- d's last rule is T-PROD-E1 or T-PROD-E2 with a product-minimal subderivation; or
- d's last rule is T-SUB with a minimal subderivation.

2.2.3 Lemma: Suppose $\Gamma \vdash e \in T$. Then:

1. There is a minimal derivation of $\Gamma \vdash e \in S$ for some S.
2. If $T = T_1 \rightarrow T_2$, then there is an arrow-minimal derivation of $\Gamma \vdash e \in S_1 \rightarrow S_2$ for some S_1 and S_2.
3. If $T = \mathit{All}\,(X{\leq}T_1)\,T_2$, then there is an All-minimal derivation of $\Gamma \vdash e \in \mathit{All}\,(X{\leq}S_1)\,S_2$ for some S_1 and S_2.
4. If $T = T_1 \times T_2$, then there is a product-minimal derivation of $\Gamma \vdash e \in S_1 \times S_2$ for some S_1 and S_2.

Proof: An execution trace of the standard type synthesis algorithm for $F_{\leq}$ [Curien and Ghelli, 1992, Cardelli *et al.*, 1994] can be viewed as a minimal derivation for a term. (Strictly speaking, an easy extension to Curien and Ghelli's algorithm is needed to handle product types.) Part (2) follows from the standard observation that, if a term e has any arrow type, then an arrow-minimal type for e may be computed from its minimal type by promoting type variables to their upper bounds until a non-variable is reached. Parts (3) and (4) are similar. □

2.2.4 Corollary: If $\Gamma \vdash e \in T$, then there is an algorithmic derivation with this conclusion.

Proof: An algorithmic derivation of any provable typing statement can be obtained from a minimal one by appending a single instance of T-SUB. □

2.2.5 Corollary: If $\Gamma \vdash e \in \mathit{All}\,(X{\leq}S_1)\,S_2$ and $\Gamma \vdash e \in \mathit{All}\,(X{\leq}T_1)\,T_2$, then $S_1 = T_1$.

Proof: By 2.2.3(3), there is an All-minimal derivation of $\Gamma \vdash e \in \mathit{All}\,(X{\leq}U_1)\,U_2$ and for some U_1 and U_2. But then, by Lemma 2.1.3, $U_1 = S_1$ and $U_1 = T_1$. □

3 Equational Theory

We present the equational theory of positive $F_{\leq}$ in two steps: first a reduction relation, then a full equational theory extending the reduction rules with more general laws for proving equivalences between programs.

3.1 Reduction

We begin with a simple set of reduction rules, capturing the intuitive operational behavior of the constructs we have introduced.

3.1.1 Definition: Single-step *reduction* on terms (at a type) is the relation generated by applying one of the following rules at any subphrase of a term (strictly speaking: at any node in a typing derivation):

$$\frac{\Gamma \vdash (\mathit{fun}\,(x{:}S)\; s)\; t \in V}{\Gamma \vdash (\mathit{fun}\,(x{:}S)\; s)\; t \triangleright [t/x]s \in V} \qquad \text{(R-Beta)}$$

$$\frac{\Gamma \vdash (\mathit{fun}\,(X{\leq}S)\; s)\; U \in V}{\Gamma \vdash (\mathit{fun}\,(X{\leq}S)\; s)\; U \triangleright [U/X]s \in V} \qquad \text{(R-Beta2)}$$

$$\frac{\Gamma \vdash \langle s_1,\, s_2\rangle.1 \in V}{\Gamma \vdash \langle s_1,\, s_2\rangle.1 \triangleright s_1 \in V} \qquad \text{(R-Prod1)}$$

$$\frac{\Gamma \vdash \langle s_1,\, s_2\rangle.2 \in V}{\Gamma \vdash \langle s_1,\, s_2\rangle.2 \triangleright s_2 \in V} \qquad \text{(R-Prod2)}$$

$$\frac{\Gamma \vdash \mathit{put}[S,\, \mathit{Top}]\; s\; t \in V}{\Gamma \vdash \mathit{put}[S,\, \mathit{Top}]\; s\; t \triangleright s \in V} \qquad \text{(R-Put-Top)}$$

$$\frac{\Gamma \vdash \mathit{put}[U{\to}S_2, U{\to}T_2]\; f\; g \in V}{\begin{array}{ll}\Gamma \vdash & \mathit{put}[U{\to}S_2, U{\to}T_2]\; f\; g \\ & \triangleright\, \mathit{fun}\,(u{:}U)\; \mathit{put}[S_2, T_2]\; (f\; u)\; (g\; u) \\ & \in V\end{array}} \qquad \text{(R-Put-Arrow)}$$

$$\frac{\Gamma \vdash \mathit{put}[\mathit{All}\,(X{\leq}U)\, S_2,\; \mathit{All}\,(X{\leq}U)\, T_2]\; f\; g \in V}{\begin{array}{ll}\Gamma \vdash & \mathit{put}[\mathit{All}\,(X{\leq}U)\, S_2,\; \mathit{All}\,(X{\leq}U)\, T_2]\; f\; g \\ & \triangleright\, \mathit{fun}\,(X{\leq}U)\; \mathit{put}[S_2, T_2]\; (f\; X)\; (g\; X) \\ & \in V\end{array}} \qquad \text{(R-Put-All)}$$

$$\frac{\Gamma \vdash \mathit{put}[S_1{\times}S_2,\; T_1{\times}T_2]\; s\; t \in V}{\begin{array}{ll}\Gamma \vdash & \mathit{put}[S_1{\times}S_2,\; T_1{\times}T_2]\; s\; t \\ & \triangleright\, \langle \mathit{put}[S_1, T_1]\; s.1\; t.1,\; \mathit{put}[S_2, T_2]\; s.2\; t.2\rangle \\ & \in V\end{array}} \qquad \text{(R-Put-Prod)}$$

The first four rules are the standard ones for functions, polymorphic functions, and projection. The others describe how various instances of *put* behave: $\mathit{put}[S,\, \mathit{Top}]$ throws away its second argument, since an element of *Top* is considered to contain no information and so the update is trivial; each of the last three pushes an instance of *put* inside of one of the other type constructors.

Note that what we have defined is a *typed* notion of reduction: types are not erased at runtime. Indeed, untyped reduction does not seem to make sense in this setting. For example,

$$put[\{x{:}Int,\ y{:}Int\}, T]\ \{x = 5,\ y = 3\}\ \{x = 4,\ y = 2\}$$

equals either $\{x = 4,\ y = 2\}$ or $\{x = 4,\ y = 3\}$, according to whether T is $\{x{:}Int,\ y{:}Int\}$ or $\{x{:}Int\}$. The typing information affects the result of the computation, and so may not be sensibly erased.

Although their formal status is just the same as the equational rules to follow, the reduction rules may be thought of as giving an abstract operational semantics to the calculus. We shall not pursue this point of view, since our main interest is in the equational theory that follows and the operational semantics in terms of untyped computations given by the per model in Section 4. However, we conjecture that the reduction relation can be shown to be strongly normalizing by a translation into pure System F (cf. [Breazu-Tannen *et al.*, 1991]). A more realistic study of operational semantics would focus on a particular evaluation strategy; the induced notion of *contextual equivalence* would then form the basis of an equational theory.

3.2 Equality

The reduction rules do not fully specify the behavior of $put[S,T]$ when S and T contain variables. For reasoning about programs, the equational theory generated by reduction is thus inadequate: it must be extended so that it embodies the assumption that such indeterminate instances of *put* will eventually be instantiated with well-behaved concrete ones.

3.2.1 Definition: *Equality* on terms (at a type) is the least reflexive, transitive, and symmetric relation closed under the following rules: R-BETA through R-PUT-PROD (replacing $\triangleright$ by $=$and changing the R- prefixes in rule names to E-), a congruence rule for each term constructor,

$$\frac{\Gamma,\ x{:}S \vdash e = e' \in T}{\Gamma \vdash \mathit{fun}\,(x{:}S)\ e = \mathit{fun}\,(x{:}S)\ e' \in S{\to}T} \qquad \text{(E-ARROW-I)}$$

$$\frac{\Gamma \vdash e_1 = e_1' \in S{\to}T \qquad \Gamma \vdash e_2 = e_2' \in S}{\Gamma \vdash e_1\ e_2 = e_1'\ e_2' \in T} \qquad \text{(E-ARROW-E)}$$

$$\frac{\Gamma,\ X{\leq}S_1 \vdash e = e' \in S_2}{\Gamma \vdash \mathit{fun}\,(X{\leq}S_1)\ e = \mathit{fun}\,(X{\leq}S_1)\ e' \in \mathit{All}\,(X{\leq}S_1)\ S_2} \qquad \text{(E-ALL-I)}$$

$$\frac{\Gamma \vdash e = e' \in \mathit{All}\,(X{\leq}S_1)\ S_2 \qquad \Gamma \vdash T \leq S_1}{\Gamma \vdash e\ T = e'\ T \in [T/X]S_2} \qquad \text{(E-ALL-E)}$$

$$\frac{\Gamma \vdash e_1 = e_1' \in S_1 \qquad \Gamma \vdash e_2 = e_2' \in S_2}{\Gamma \vdash \langle e_1,\ e_2\rangle = \langle e_1',\ e_2'\rangle \in S_1{\times}S_2} \qquad \text{(E-PROD-I)}$$

$$\frac{\Gamma \vdash e = e' \in S_1{\times}S_2}{\Gamma \vdash e.i = e'.i \in S_i \quad \text{for } i \in \{1,2\}} \qquad \text{(E-PROD-E)}$$

η-conversion for both term and type applications,

$$\frac{\Gamma \vdash \mathit{fun}\,(x{:}S)\ e\ x \in T \qquad x \text{ not free in } e}{\Gamma \vdash \mathit{fun}\,(x{:}S)\ e\ x = e \in T} \qquad \text{(E-ETA)}$$

$$\frac{\Gamma \vdash \mathit{fun}\,(X{\leq}S)\; e\; X \in T \qquad X \text{ not free in } e}{\Gamma \vdash \mathit{fun}\,(X{\leq}S)\; e\; X = e \in T} \tag{E-ETA2}$$

η-conversion (surjective pairing) for products,

$$\frac{\Gamma \vdash e \in T_1{\times}T_2}{\Gamma \vdash \langle e.1,\; e.2\rangle = e \in T_1{\times}T_2} \tag{E-SURJ}$$

subsumption,

$$\frac{\Gamma \vdash e = e' \in S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash e = e' \in T} \tag{E-SUB}$$

identification of all elements of Top,

$$\frac{\Gamma \vdash e \in \mathit{Top} \qquad \Gamma \vdash e' \in \mathit{Top}}{\Gamma \vdash e = e' \in \mathit{Top}} \tag{E-TOP}$$

a rule describing the behavior of *put* on variables,

$$\frac{\Gamma \vdash \mathit{put}[X, \Gamma(X)]\; s\; (\mathit{put}[\Gamma(X), U]\; s\; t) \in T}{\Gamma \;\vdash \mathit{put}[X, U]\; s\; t = \mathit{put}[X, \Gamma(X)]\; s\; (\mathit{put}[\Gamma(X), U]\; s\; t) \in T} \tag{E-PUT-VAR}$$

and three rules characterizing the behavior of *put* uniformly at all types:

$$\frac{\Gamma \vdash \mathit{put}[S, T]\; s\; t \in T}{\Gamma \vdash \mathit{put}[S, T]\; s\; t = t \in T} \tag{E-PUT-1}$$

$$\frac{\Gamma \vdash \mathit{put}[S, T]\; s\; s \in S}{\Gamma \vdash \mathit{put}[S, T]\; s\; s = s \in S} \tag{E-PUT-2}$$

$$\frac{\Gamma \vdash \mathit{put}[S, T]\; (\mathit{put}[S, T]\; s\; t')\; t \in S}{\Gamma \vdash \mathit{put}[S, T]\; (\mathit{put}[S, T]\; s\; t')\; t = \mathit{put}[S, T]\; s\; t \in S} \tag{E-PUT-3}$$

The most interesting of these laws are the last four. E-PUT-VAR is an instance of a general transitivity principle (cf. 3.3.1); intuitively, it permits the decomposition of an unknown *put*-function (one whose lower type is a variable) into a "smaller" unknown *put*-function, relating the variable to its bound, plus a known *put*-function relating the bound to the upper type. E-PUT-1 through E-PUT-3 capture the intuition that *put*-functions perform updates. Indeed, these laws are complete for this intuition, in a sense that will be made precise in the development of our semantic model (Proposition 4.2.4).

It is easy to verify the following simple sanity check of the equational theory.

3.2.2 Fact: If $\Gamma \vdash e = e' \in T$, then $\Gamma \vdash e \in T$ and $\Gamma \vdash e' \in T$.

3.3 Derived Laws

A number of additional laws for manipulating *put* may be derived from the equational theory. For example, the following rule expresses the intuition that $put[S,T]$ acts like an identity outside of the T part of S:

$$s = put[S,T]\ (put[S,T]\ s\ t')\ s \in S.$$

This is a straightforward consequence of E-PUT-2 and E-PUT-3. The next proposition gives some more interesting derived rules.

3.3.1 Proposition: The following general laws of reflexivity and transitivity for *put*, analogous to the general laws of reflexivity and transitivity of subtyping, may be derived.

$$\frac{\Gamma \vdash put[S,S]\ s\ t \in S}{\Gamma \vdash put[S,S]\ s\ t = t \in S} \qquad \text{(E-PUT-REFL)}$$

$$\frac{\Gamma \vdash put[S,R]\ s\ (put[R,T]\ s\ t) \in S}{\Gamma \vdash put[S,T]\ s\ t = put[S,R]\ s\ (put[R,T]\ s\ t) \in S} \qquad \text{(E-PUT-TRANS)}$$

Proof: E-PUT-REFL is a special case of E-PUT-1. For transitivity, first observe that by the form of the typing rules, the premise implies that the *put*-subphrases are well typed, which in turn implies $\Gamma \vdash S \leq R$ and $\Gamma \vdash R \leq T$. Now proceed by induction on algorithmic derivations of $\Gamma \vdash S \leq R$ and $\Gamma \vdash R \leq T$.

- Case $T = Top$. By E-PUT-TOP, the left-hand side is equal to s and the right-hand side is equal to $put[S,R]\ s\ s \in S$. These are equal by E-PUT-2.
- Case $R = Top$. Then T must be Top and we have an instance of the previous case.
- Case $S = X$. There are two subcases to consider. If $R = X$, then use E-PUT-REFL applied to $put[S,R]$. Otherwise, we have $\Gamma \vdash \Gamma(X) \leq R$ by assumption. We then calculate as follows:

$$\begin{array}{lll} & put[X,T]\ s\ t & \\ = & put[X,\Gamma(X)]\ s\ (put[\Gamma(X),T]\ s\ t) & \text{by E-PUT-VAR} \\ = & put[X,\Gamma(X)]\ s\ (put[\Gamma(X),R]\ s\ (put[R,T]\ s\ t)) & \text{by the induction hypothesis (IH)} \\ = & put[X,R]\ s\ (put[R,T]\ s\ t) & \text{by E-PUT-VAR.} \end{array}$$

- Case $S = All\,(X{\leq}U)\,S_2$ and $R = All\,(X{\leq}U)\,R_2$ and $T = All\,(X{\leq}U)\,T_2$, with $\Gamma,\ X{\leq}U \vdash S_2 \leq R_2 \leq T_2$. By the IH and straightforward equational reasoning.
- The cases for arrow and product are similar. □

3.3.2 Corollary: If $s \in S$, $r \in R$, and $t \in T$, with $S \leq R \leq T$, then

$$put[S,T]\ (put[S,R]\ s\ r)\ t = put[S,R]\ s(put[R,T]\ r\ t).$$

Proof: Apply E-PUT-TRANS to the left-hand side and reduce the result by using E-PUT-3 followed by E-PUT-1. □

We shall also need the simple observation that all of the judgements of this system (typing, subtyping, and equality) are stable under substitution.

3.3.3 Lemma:

1. If $\Gamma,\ x{:}S,\ \Delta \vdash J$ for some judgement J and $\Gamma \vdash e \in S$, then $\Gamma,\ \Delta \vdash [e/x]J$.
2. If $\Gamma,\ X{\leq}U,\ \Delta \vdash J$ and $\Gamma \vdash S{\leq}U$, then $\Gamma,\ [S/X]\Delta \vdash [S/X]J$.

Proof: By induction on derivations. The only interesting cases are the rules that contain free variables: S-TVAR, T-VAR, and E-PUT-VAR. The cases for S-TVAR and T-VAR follow directly from the assumptions in parts (1) and (2).

Suppose the last step in a derivation is an instance of E-PUT-VAR:

$$\frac{\Gamma,\ X{\leq}U,\ \Delta \vdash put[X,U]\ s\ (put[U,V]\ s\ t) \in T}{\Gamma,\ X{\leq}U,\ \Delta \vdash put[X,V]\ s\ t = put[X,U]\ s(put[U,V]\ s\ t) \in T}$$

Using the induction hypothesis on the premise, we obtain $\Gamma,\ [S/X]\Delta \vdash put[S,U]\ [S/X]s\ (put[U,[S/X]V]\ [S/X]s\ [S/X]t) \in [S/X]T$, from which the desired conclusion follows by E-PUT-TRANS. □

3.4 Non-Overlapping Updates

In practical situations, we are often interested in updating non-overlapping portions of a complex data structure. In particular, Section 5 will introduce an encoding of record types in terms of the built-in products and *Top*. For these record types, the natural intuition "non-overlapping portions = disjoint sets of fields" will arise from the following definition of non-overlap for our basic types.

3.4.1 Definition: We say that two types S and T are *non-overlapping*, written $S \perp T$, if

- either S or T is *Top*; or
- $S = S_1 \times S_2$ and $T = T_1 \times T_2$ with $S_1 \perp T_1$ and $S_2 \perp T_2$; or
- $S = S_1 {\rightarrow} S_2$ and $T = T_1 {\rightarrow} T_2$ with $S_2 \perp T_2$; or
- $S = All\,(A{\leq}S_1)\,S_2$ and $T = All\,(A{\leq}T_1)\,T_2$ with $S_2 \perp T_2$.

Clearly, if $S \perp T$, then $T \perp S$. Also, if $S \perp T$ and S is a variable, then $T = Top$. More interestingly:

3.4.2 Lemma: If $S \perp T$ and $\Gamma \vdash S \leq U$, then $U \perp T$.

Proof: By straightforward induction on an algorithmic derivation of $\Gamma \vdash S \leq U$. If S is a variable, then U is *Top* and the result is immediate; if S is not a variable and U is not *Top*, then by the definition of algorithmic derivations, S and U have the same outermost constructor. By the definition of non-overlapping types, T must either be *Top* or share the same outermost constructor as S and U. The *Top* case is immediate; the other proceeds by induction. □

The motivating feature of non-overlapping types is that updates to non-overlapping portions of a complex data structure do not interfere:

3.4.3 Proposition: Suppose that $S \perp T$ and let U be a common subtype of S and T in some context Γ. Then for all $s \in S$ and $u \in U$,

$$put[U,S]\ u\ s = u \in T.$$

Proof: By simultaneous induction on a pair of algorithmic derivations of $\Gamma \vdash U \leq S$ and $\Gamma \vdash U \leq T$.

If S is *Top*, then the result follows by E-PUT-TOP and E-SUB. If T is *Top*, then the result is an instance of E-TOP.

If $U = X$ for some variable X, then, since we have already dealt with the cases where either S or T is *Top*, neither S nor T can be a variable (in view of the above observation on overlapping variables); we must therefore have subderivations $\Gamma \vdash \Gamma(X) \leq S$ and $\Gamma \vdash \Gamma(X) \leq T$. By E-PUT-1,

$$put[X, \Gamma(X)]\ u\ (put[\Gamma(X), S]\ u\ s) = put[\Gamma(X), S]\ u\ s \in \Gamma(X).$$

By E-PUT-TRANS (on the left-hand side) and E-SUB, this becomes

$$put[X, S]\ u\ s = put[\Gamma(X), S]\ u\ s \in T.$$

Now, using the induction hypothesis, the right-hand side equals u in T, from which the result follows by the transitivity of equality.

The remaining structural cases follow by one of the rules E-PUT-ARROW, E-PUT-ALL, or E-PUT-PROD, the induction hypothesis, and one E-ETA, E-ETA2, and E-SURJ. Notice, in the arrow (resp. quantifier) case, that U being a lower bound implies that the domains (bounds) of S and T are identical. $\square$

4 Realizability Semantics

In this section we define a partial equivalence relation model for positive $F_{\leq}$. This establishes consistency of the equational theory in Section 3 and also lays down the intended semantics of positive $F_{\leq}$ in terms of untyped computation. As usual, types are interpreted as partial equivalence relations (pers) on the natural numbers and terms are interpreted as numbers denoting partial recursive algorithms. The new feature of our model is the interpretation of subtyping judgements. We define a notion of *updateable subper* extending the usual notion of "subper = set-inclusion" with an extra map satisfying the three put-laws from Section 3. The translation from typed source-language terms to elements of the model is not a simple erasure of types; rather, subtyping judgements are translated into "realizers" for the corresponding *put*-functions. In particular, this means that the type annotations in a term affect its meaning.

We prove that a per S is an updateable subper of T iff S admits a decomposition into $S \cong T \times R$ for some per R, thus establishing the correspondence of positive $F_{\leq}$ with the intuitive understanding of updating. (Oles [1985] establishes an analogous property for his state-space semantics.)

4.1 Preliminaries

We assume some coding of partial recursive functions as natural numbers. The application of the mth partial recursive function to argument n is written $m \cdot n$. Application associates to the left, so $m \cdot n \cdot k = (m \cdot n) \cdot k$. (If $m \cdot n$ is undefined, then so is $m \cdot n \cdot k$; in general, we adopt the convention that any expression with an undefined subexpression is itself undefined.) We assume some recursive encoding of pairing, writing $\langle m, n \rangle$ for pairing and π_1, π_2 for the projections. Thus for $m, n \in \omega$ we have $\pi_1(\langle m, n \rangle) = m$, $\pi_2(\langle m, n \rangle) = n$, and $\langle \pi_1(m), \pi_2(m) \rangle = m$. We also make use of semantic abstraction on recursive functions: if $f(x)$ is some description of a partial recursive function with input x, then we write $\lambda x.f(x)$ for the corresponding code. From the definition, we immediately have the "semantic β-rule"

$$(\lambda x.\ f(x)) \cdot m = f(m) \qquad \text{(SEMANTIC BETA)}$$

but not necessarily the "semantic η-rule" $\lambda x.\ (m \cdot x) = m$.

A *partial equivalence relation* (per) is a symmetric and transitive relation on the set ω of natural numbers. If A is a per we write $m\ \{A\}\ n$ if m and n are related by A; we also write $m \sim n$ in this situation if A is clear from the context. We write $\mathrm{dom}(A)$ for the set $\{n \,|\, n\ \{A\}\ n\}$. Clearly the restriction of A to $\mathrm{dom}(A)$ is an equivalence relation and $m\ \{A\}\ n$ implies $m, n \in \mathrm{dom}(A)$. The *product* and *exponential* of pers A and B are the pers defined by

$$\begin{aligned}
&m\ \{A \times B\}\ n \text{ iff } \pi_1(m)\ \{A\}\ \pi_1(n) \wedge \pi_2(m)\ \{B\}\ \pi_2(n)\\
&m\ \{A{\rightarrow}B\}\ n \text{ iff } \forall a, a'.\ a\ \{A\}\ a' \Rightarrow m \cdot a\ \{B\}\ n \cdot a'.
\end{aligned}$$

In the last clause, $m \cdot a\ \{B\}\ n \cdot a'$ means that both computations are defined and their results are related in B.

A *morphism* from per A to per B is an equivalence class of elements of $\mathrm{dom}(A{\rightarrow}B)$, two numbers m and n being identified if $m\ \{A{\rightarrow}B\}\ n$. We often identify notationally a morphism and a representative (code) for it. The pers together with their morphisms form a category, which is cartesian closed. Product and exponential are given as above; the per $\mathit{Top} = \omega \times \omega$ is a terminal object. A morphism is an *inclusion* if it is coded by the identity function. There exists an inclusion from per A to B iff $A \subseteq B$ *qua* sets of pairs. In this case we say that A is a *subper* of B.

4.1.1 Fact: Let A and B be pers and $m, n \in \mathrm{dom}(A{\rightarrow}B)$. Then

$$m\ \{A{\rightarrow}B\}\ n \quad \text{iff} \quad \forall x{\in}\mathrm{dom}(A).\ m \cdot x\ \{B\}\ n \cdot x.$$

For more information on standard properties of pers, see [Gunter, 1992].

4.2 The Interpretation of Subtyping

4.2.2 Definition: We say that A is an *updateable subper* of B and write $A \leq B$, if A is a subper of B and there exists a morphism $p \in A \rightarrow (B{\rightarrow}A)$ satisfying the three put-equations from 3.2.1. More precisely, $A \leq B$ if $A \subseteq B$ and for some code $p \in \omega$ we have:

$$\begin{array}{lll}
0) & \forall a, a', b, b'\ .\ a\ \{A\}\ a' \Rightarrow b\ \{B\}\ b' \Rightarrow (p \cdot a \cdot b\ \{A\}\ p \cdot a' \cdot b') & \\
1) & \forall a \in \mathrm{dom}(A), b \in \mathrm{dom}(B)\ .\ p \cdot a \cdot b\ \{B\}\ b & \text{(PUT1)}\\
2) & \forall a \in \mathrm{dom}(A)\ .\ p \cdot a \cdot a\ \{A\}\ a & \text{(PUT2)}\\
3) & \forall a \in \mathrm{dom}(A), b, b' \in \mathrm{dom}(B)\ .\ p \cdot (p \cdot a \cdot b') \cdot b\ \{A\}\ p \cdot a \cdot b & \text{(PUT3)}
\end{array}$$

We then say that p *witnesses* $A \leq B$, written $p \models A \leq B$.

4.2.3 Remark: The witness p for $A \leq B$ is not necessarily unique. For if φ is an isomorphism when viewed as an element of $A{\rightarrow}A$ and is the inclusion map when viewed as an element of $A{\rightarrow}B$, then $\lambda s.\ \lambda t.\ \varphi^{-1} \cdot (p \cdot (\varphi \cdot s) \cdot t)$ also witnesses $A \leq B$, where φ^{-1} is a code for the inverse of φ.

4.2.4 Proposition: Let A be an updateable subper of B. Then there exists a per R such that A is isomorphic to $B \times R$. Moreover, if p witnesses $A \leq B$, then the isomorphism can be chosen in such a way that p arises canonically. This means that if $\varphi : A \cong B \times R$ is the isomorphism, then $\pi_1 \circ \varphi : A{\rightarrow}B$ is an inclusion and the morphism coded by p equals

$$\lambda a.\ \lambda b.\ \varphi^{-1}(\langle b, \pi_2(\varphi(a))\rangle).$$

Proof: Assume $p \models A \leq B$. The idea is that if A admits a decomposition into $B \times R$ then every element of R should give rise to a function f from B to A with the property that the behavior of f can be recovered (by updating) from the single instance $f(b)$ for an arbitrary $b \in \mathrm{dom}(B)$; that is, $f(b') = p \cdot f(b) \cdot b'$ for all $b, b' \in \mathrm{dom}(B)$. Indeed, we can just take the set of such functions as a candidate for R. More precisely, define R by

$$f \,\{R\}\, f' \text{ iff } f \,\{B \to A\}\, f' \wedge \forall b, b' \in \mathrm{dom}(B) \,.\, f \cdot b' \,\{A\}\, p \cdot (f \cdot b) \cdot b'.$$

Notice that R is a per and that $R \subseteq B \to A$.

We claim that $\varphi = \lambda a.\ \langle a, \lambda b.\ p \cdot a \cdot b\rangle$ codes an isomorphism from A to $B \times R$. So first we must show that φ is at least a morphism from A to $B \times R$. Since it is clearly a morphism from A to $B \times (B \to A)$ it remains to show that

$$\forall b, b' \in \mathrm{dom}(B) \,.\, p \cdot a \cdot b' \,\{A\}\, p \cdot (p \cdot a \cdot b) \cdot b',$$

but this is an instance of PUT3. Next, we claim that application, i.e. $\psi = \lambda\langle b, f\rangle.\ f \cdot b$, is the required inverse of φ. Indeed, if $a \in \mathrm{dom}(A)$ then

$$\begin{array}{lll} & \psi \cdot (\varphi \cdot a) & \\ = & p \cdot a \cdot a & \\ \{A\} & a & \text{by PUT2,} \end{array}$$

and, conversely, if $\langle b, f\rangle \in \mathrm{dom}(B \times R)$ then

$$\begin{array}{lll} & \pi_1(\varphi \cdot (\psi \cdot \langle b, f\rangle)) & \\ = & f \cdot b & \\ \{A\} & p \cdot (f \cdot b) \cdot b & \text{since } f \in \mathrm{dom}(R) \\ \{B\} & b & \text{by PUT1,} \end{array}$$

and also

$$\begin{array}{lll} & \pi_2(\varphi \cdot (\psi \cdot \langle b, f\rangle)) & \\ = & \lambda b'.\, p \cdot (f \cdot b) \cdot b' & \\ \{B \to A\} & \lambda b'.\, f \cdot b' & \text{since } f \in \mathrm{dom}(R) \\ \{B \to A\} & f, & \end{array}$$

as required. Finally, the first component of φ is coded by the identity and is thus an inclusion; the fact that p equals $\lambda a.\ \lambda b.\ \varphi^{-1}(\langle b, \pi_2(\varphi(a))\rangle)$ may be checked by expanding definitions. □

We have seen that equations PUT1 to PUT3 are complete in the sense that they characterize the canonical situation where the subtype coercion is a product projection and *put* replaces one product component and leaves the other one unchanged. Since, in view of remark 4.2.3, a witness for $A \leq B$ is not uniquely determined by the mere fact that $A \subseteq B$, we must give a "proof-relevant" interpretation of the subtyping judgement, specifying the particular witness chosen. Clearly, for the type constructors a canonical witness can be defined according to the equations in Section 3.2, but for type variables there is no canonical way to choose such a witness; we therefore require that the environment supply a witness for each assumption $X \leq U$.

4.3 The Model

A *type environment* is a finite function from type variables to pers. If τ is a type environment then for $X \in \mathrm{dom}(\tau)$ the per $\tau(X)$ is meant to interpret the type X. A *term environment* is a finite function from term *and* type variables to natural numbers. We use the notation $\tau[X \mapsto R]$ for updating a finite function at one point. If η is a term environment and x is a term variable in the domain of η, then $\eta(x)$ is meant to interpret the term x, while for a type variable $X \in \mathrm{dom}(\eta)$ the code $\eta(X)$ is meant to interpret the put-function relating the type X to its upper bound. An *environment* is a pair (τ, η) of a type environment and a term environment.

If T is a type expression and τ a type environment with $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\tau)$, then the *interpretation of T under τ*, written $[\![T]\!]_\tau$, is the per defined as follows:

$$
\begin{array}{lcl}
[\![X]\!]_\tau & = & \tau(X) \\
[\![\mathit{Top}]\!]_\tau & = & \mathit{Top} \\
[\![T_1 \to T_2]\!]_\tau & = & [\![T_1]\!]_\tau \to [\![T_2]\!]_\tau \\
[\![T_1 \times T_2]\!]_\tau & = & [\![T_1]\!]_\tau \times [\![T_2]\!]_\tau \\
m \; \{[\![\mathit{All}\,(X {\leq} U)\, T]\!]_\tau\} \; n & \iff & \begin{array}{l} \forall R \in \mathrm{PER}, p \in \omega. \\ \quad p \models R \leq [\![U]\!]_\tau \Rightarrow m \cdot p \; \{[\![T]\!]_{\tau[X \mapsto R]}\} \; n \cdot p. \end{array}
\end{array}
$$

(It is easy to check that the relation defined by the last clause is symmetric and transitive; recall that $m \cdot p \; \{[\![T]\!]_{\tau[X \mapsto R]}\} \; n \cdot p$ means in particular that both sides are defined.)

4.3.5 Lemma: Let τ be a type environment such that $\mathrm{FTV}([T/X]S) \subseteq \mathrm{dom}(\tau)$. Then $[\![[T/X]S]\!]_\tau = [\![S]\!]_{\tau[X \mapsto [\![T]\!]_\tau]}$.

Proof: By induction on the structure of S. □

Let η be a term environment, Γ a context, and S, T type expressions. The (possibly undefined) *witness associated to S and T* under Γ and η, written $\mathrm{Put}_{\Gamma;\eta}[S, T]$, is the natural number defined as follows:

$$
\begin{array}{l}
\mathrm{Put}_{\Gamma;\eta}[S, \mathit{Top}] = \\
\quad \lambda s.\ \lambda t.\ s \\
\mathrm{Put}_{\Gamma;\eta}[X, X] = \\
\quad \lambda x.\ \lambda x'.\ x' \\
\mathrm{Put}_{\Gamma;\eta}[X, T] = \\
\quad \lambda x.\ \lambda t.\ \eta(X) \cdot x \cdot (\mathrm{Put}_{\Gamma;\eta}[\Gamma(X), T] \cdot x \cdot t) \qquad (\text{when } T \neq X) \\
\mathrm{Put}_{\Gamma;\eta}[U \to S, U \to T] = \\
\quad \lambda s.\ \lambda t.\ \lambda u.\ \mathrm{Put}_{\Gamma;\eta}[S, T] \cdot (s \cdot u) \cdot (t \cdot u) \\
\mathrm{Put}_{\Gamma;\eta}[S_1 \times S_2, T_1 \times T_2] = \\
\quad \lambda s.\ \lambda t.\ \langle \mathrm{Put}_{\Gamma;\eta}[S_1, T_1] \cdot \pi_1(s) \cdot \pi_1(t),\ \mathrm{Put}_{\Gamma;\eta}[S_2, T_2] \cdot \pi_2(s) \cdot \pi_2(t) \rangle \\
\mathrm{Put}_{\Gamma;\eta}[\mathit{All}\,(X {\leq} U)\, S, \mathit{All}\,(X {\leq} U)\, T] = \\
\quad \lambda s.\ \lambda t.\ \lambda p.\ \mathrm{Put}_{\Gamma, X \leq U; \eta[X \mapsto p]}[S, T] \cdot (s \cdot p) \cdot (t \cdot p) \\
\text{undefined otherwise.}
\end{array}
$$

Note the similarity between the clauses for function types and polymorphic types; in the latter, the argument p stands for a witness associated to the subtyping assumption $X \leq U$. The clause for $\mathrm{Put}_{\Gamma;\eta}[X, T]$ implements the equational rule E-PUT-VAR.

4.3.6 Fact: Let Γ be a context, S and T type expressions, and η_1 and η_2 term environments that agree on all type variables. Then $\mathrm{Put}_{\Gamma;\eta_1}[S, T]$ equals $\mathrm{Put}_{\Gamma;\eta_2}[S, T]$ if either is defined (since the definition of Put does not look at term variable bindings).

We say that an environment (η, τ) *satisfies* a context Γ, written $(\eta, \tau) \models \Gamma$, if: (1) η is defined on all term and type variables occurring in Γ; (2) τ is defined on all type variables occurring in Γ; (3) whenever Γ admits a decomposition $\Gamma = \Gamma_1, X{\leq}T, \Gamma_2$, we have $(\eta, \tau) \models \Gamma_1$ and $\eta(X) \models \tau(X) \leq [\![\Gamma(X)]\!]_\tau$; and (4) for each $x \in FV(\Gamma)$, we have $\eta(x) \in \mathrm{dom}([\![\Gamma(x)]\!]_\tau)$.

4.3.7 Lemma [Reflexivity of Put]: If S is a type in a context Γ and $(\eta, \tau) \models \Gamma$, then

$$\mathrm{Put}_{\Gamma;\eta}[S,S] \quad \{[\![S{\to}S{\to}S]\!]_\tau\} \quad \lambda x.\ \lambda y.\ y.$$

Proof: By induction on the structure of S. □

4.3.8 Theorem [Soundness of subtyping]: If $\Gamma \vdash S \leq T$ and $(\eta, \tau) \models \Gamma$ then

$$\mathrm{Put}_{\Gamma;\eta}[S,T] \models [\![S]\!]_\tau \leq [\![T]\!]_\tau.$$

Proof: Proceed by induction on an algorithmic derivation of $\Gamma \vdash S \leq T$. We show just the cases for variables and universal quantification.

Let $S = X$ and $T = \Gamma(X)$. Now

$$\begin{array}{rll} & \mathrm{Put}_{\Gamma;\eta}[S,T] & \\ = & \lambda x.\ \lambda t.\ \eta(X) \cdot x \cdot (\mathrm{Put}_{\Gamma;\eta}[\Gamma(X), \Gamma(X)] \cdot x \cdot t) & \\ \sim & \lambda x.\ \lambda t.\ \eta(X) \cdot x \cdot t & \text{by 4.3.7} \\ \sim & \eta(X), & \end{array}$$

so the result follows by the assumption on η and τ. If the last rule used in the derivation of $S \leq T$ was either reflexivity on a variable or transitivity with a variable assumption as its left-hand premise, we obtain the result by straightforward equality reasoning from the definition and the induction hypothesis.

Now let us consider the case of universal quantification, where $S = \mathit{All}\,(X{\leq}U)\,S'$ and $T = \mathit{All}\,(X{\leq}U)\,T'$. We first have to show that $\mathrm{Put}_{\Gamma;\eta}[S,T]$ codes a morphism between the appropriate types. Let $s, s', t, t' \in \omega$ be such that $s\ \{S\}\ s'$ and $t\ \{T\}\ t'$. Furthermore let $R \in \mathrm{PER}$ and $p \in \omega$ be such that $p \models R \leq [\![U]\!]_\tau$. We must show that

$$\mathrm{Put}_{\Gamma;\eta}[S,T] \cdot s \cdot t \cdot p\ \{[\![S]\!]_{\tau[X \mapsto R]}\}\ \mathrm{Put}_{\Gamma;\eta}[S,T] \cdot s' \cdot t' \cdot p.$$

Unfolding the induction definition, this becomes

$$\mathrm{Put}_{\Gamma,X\leq U;\eta[X\mapsto p]}[S',T'] \cdot (s \cdot p) \cdot (t \cdot p)\ \{[\![S]\!]_{\tau[X\mapsto R]}\}\ \mathrm{Put}_{\Gamma,X\leq U;\eta[X\mapsto p]}[S',T'] \cdot (s' \cdot p) \cdot (t' \cdot p).$$

Now, $(\eta[X \mapsto p], \tau[X \mapsto R])$ satisfies the context $\Gamma, X{\leq}U$, so the result follows from the induction hypothesis used with the extended environment. The other three clauses of the definition of $A \leq B$ follow similarly. □

4.3.9 Lemma [Transitivity of Put]: If $\Gamma \vdash S \leq T \leq U$ and $(\eta, \tau) \models \Gamma$, then

$$\mathrm{Put}_{\Gamma;\eta}[S,U] \quad \{[\![S{\to}U{\to}S]\!]_\tau\} \quad \lambda s.\ \lambda u.\ \mathrm{Put}_{\Gamma;\eta}[S,T] \cdot s \cdot (\mathrm{Put}_{\Gamma;\eta}[T,U] \cdot s \cdot u).$$

Proof: By induction on a pair of algorithmic derivations of $\Gamma \vdash S \leq T$ and $\Gamma \vdash T \leq U$, following the same pattern as the proof of Lemma 3.3.1. □

4.3.10 Lemma [Weakening of Put]: If S and T are types in Γ and Γ, Δ is any extension of Γ, then

$$\mathrm{Put}_{\Gamma;\eta}[S,T] = \mathrm{Put}_{\Gamma,\,\Delta;\eta}[S,T].$$

Proof: By inspection of the definition of Put. $\Box$

4.3.11 Lemma [Substitutivity of Put]: Suppose

$$\begin{array}{l} \Gamma \vdash S \leq U \\ (\eta,\,\tau) \models \Gamma,\, X{\leq}U,\, \Delta \\ \tau(X) = [\![S]\!]_\tau \\ \eta(X) = \mathrm{Put}_{\Gamma;\eta}[S,U] \\ \Gamma,\, X{\leq}U,\, \Delta \vdash P \leq Q. \end{array}$$

Then

$$\mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[P,Q] \quad \{[\![P \to Q \to P]\!]_\tau\} \quad \mathrm{Put}_{\Gamma,\,[S/X]\Delta;\eta}[[S/X]P,[S/X]Q].$$

Proof: Notice first (by an easy induction on the length of Δ, using Lemma 4.3.5) that $(\eta,\,\tau) \models \Gamma,\,[S/X]\Delta$. Moreover, by the soundness of subtyping,

$$\mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[P,Q] \in \mathrm{dom}([\![P \to Q \to P]\!]_\tau).$$

By the the stability of subtyping under substitution (3.3.3), we have $\Gamma,\,[S/X]\Delta \vdash [S/X]P \leq [S/X]Q$. Therefore, by the soundness of subtyping,

$$\mathrm{Put}_{\Gamma,\,[S/X]\Delta;\eta}[[S/X]P,[S/X]Q] \in \mathrm{dom}([\![[S/X]P \to [S/X]Q \to [S/X]P]\!]_\tau),$$

and, since $\tau = \tau[X \mapsto [\![S]\!]_\tau]$, Lemma 4.3.5 yields

$$[\![P \to Q \to P]\!]_\tau \quad = \quad [\![[S/X]P \to [S/X]Q \to [S/X]P]\!]_\tau.$$

Now proceed by induction on an algorithmic derivation of $\Gamma \vdash P \leq Q$.

- Case $Q = \mathit{Top}$. Then the right- and left-hand sides expand to identical terms.
- Case $P = Q = X$. Then

$$\begin{array}{rll} & \mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[P,Q] & \\ = & \lambda x.\ \lambda x'.\ x & \text{by definition} \\ \{[\![S{\to}S{\to}S]\!]_\tau\} & \mathrm{Put}_{\Gamma,\,[S/X]\Delta;\eta}[S,S] & \text{by 4.3.7} \\ = & \mathrm{Put}_{\Gamma,\,[S/X]\Delta;\eta}[[S/X]P,[S/X]Q]. & \end{array}$$

- Case $P = X$ and $Q \neq X$. Then

$$\begin{array}{rll} & \mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[P,Q] & \\ = & \lambda x.\ \lambda q.\ \eta(X) \cdot x \cdot (\mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[U,Q] \cdot x \cdot q) & \text{by definition} \\ = & \lambda x.\ \lambda q.\ \mathrm{Put}_{\Gamma;\eta}[S,U] \cdot x \cdot (\mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[U,Q] \cdot x \cdot q) & \text{by assumption} \\ \sim & \mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[S,Q] & \text{by 4.3.10 and 4.3.9} \\ \sim & \mathrm{Put}_{\Gamma,\,[S/X]\Delta;\eta}[S,[S/X]Q] & \text{by the induction hypothesis.} \end{array}$$

- Case $P = Y$ and $Q = Y$ for some variable $Y \neq X$. Then both sides expand to identical terms.

- Case $P = Y$ for some variable $Y \neq X$ and $Q \neq Y$. The result follows by unfolding the definitions and using the induction hypothesis, observing that $[S/X]((\Gamma,\ X{\leq}U,\ \Delta)(Y))$ equals $(\Gamma,\ [S/X]\Delta)(Y)$.

- $P = V{\rightarrow}P'$ and $Q = V{\rightarrow}Q'$. Then

$$\begin{array}{lll} & \mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[P,Q] & \\ = & \lambda p.\ \lambda q.\ \lambda v.\ \mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[P',Q'] \cdot (p \cdot v) \cdot (q \cdot v) & \text{by definition} \\ \sim & \lambda p.\ \lambda q.\ \lambda v.\ \mathrm{Put}_{\Gamma,\,[S/X]\Delta;\eta}[[S/X]P',[S/X]Q'] \cdot (p \cdot v) \cdot (q \cdot v) & \text{by the IH} \\ = & \mathrm{Put}_{\Gamma,\,[S/X]\Delta;\eta}[[S/X]P,[S/X]Q]. & \end{array}$$

- Case $P = P_1{\times}P_2$ and $Q = Q_1{\times}Q_2$. Similar.

- Case $P = \mathit{All}\,(Y{\leq}V)\,P'$ and $Q = \mathit{All}(Y{\leq}V)Q'$. After expanding the definitions and applying Fact 4.1.1, we must show

$$\begin{array}{c} \mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta,\,Y\leq V;\eta[Y\mapsto p]}[P',Q'] \cdot (s \cdot p) \cdot (t \cdot p) \\ \{[\![P']\!]_{\tau[Y\mapsto R]}\} \\ \mathrm{Put}_{\Gamma,\,[S/X]\Delta,\,Y\leq[S/X]V;\eta[Y\mapsto p]}[[S/X]P',[S/X]Q'] \cdot (s \cdot p) \cdot (t \cdot p) \end{array}$$

  for all $s \in \mathrm{dom}([\![P]\!]_\tau)$, $t \in \mathrm{dom}([\![Q]\!]_\tau)$, and p, R such that $p \models R \leq [\![V]\!]_\tau$. But this is an instance of the induction hypothesis, since $(\eta[Y \mapsto p],\ \tau[Y \mapsto R]) \models \Gamma,\ X{\leq}U,\ \Delta,\ Y{\leq}V$. $\square$

Our aim is to give meaning to raw terms rather than typing derivations in order to avoid coherence considerations (cf. [Breazu-Tannen *et al.*, 1991]). In particular, the interpretation should be independent of particular instances of the subsumption rule. But since the interpretation of $put[S,T]$ via the Put function is context dependent, we do have to make contexts part of the interpretation of terms. Moreover in the interpretation of a polymorphic application $e\,S$ we want to apply the meaning of e to the function $\mathrm{Put}_{\Gamma;\eta}[S,U]$, where U is the upper bound of the universally quantified type of e. So we need *some* typing information on e. Fortunately, our subtyping system is such that the upper bound U is uniquely determined by the term e and the context Γ.

4.3.12 Lemma: There is a partial function $\mathrm{Bound}_\Gamma(e)$ assigning types to terms in a given context, with the property that if $\Gamma \vdash e \in \mathit{All}\,(X{\leq}U)\,S$ then $\mathrm{Bound}_\Gamma(e) = U$. Moreover, there is an algorithm that calculates $\mathrm{Bound}_\Gamma(e)$ whenever it is defined.

Proof: Immediate from Corollary 2.2.5 and the fact that there exists an algorithm for computing an All-minimal type for any term with a quantified type. $\square$

Now we are ready to define the interpretation of terms. Let η be a term environment, Γ a context, and e a term. The (possibly undefined) interpretation of e under Γ and η is the natural number given by the following clauses:

$$\begin{array}{lcl}
[\![x]\!]_{\Gamma;\eta} & = & \eta(x) \\
[\![e.1]\!]_{\Gamma;\eta} & = & \pi_1([\![e]\!]_{\Gamma;\eta}) \\
[\![e.2]\!]_{\Gamma;\eta} & = & \pi_2([\![e]\!]_{\Gamma;\eta}) \\
[\![\langle e_1,\, e_2\rangle]\!]_{\Gamma;\eta} & = & \langle [\![e_1]\!]_{\Gamma;\eta}, [\![e_2]\!]_{\Gamma;\eta}\rangle \\
[\![\mathit{fun}\,(x{:}T)\; e]\!]_{\Gamma;\eta} & = & \lambda m.\; [\![e]\!]_{\Gamma,x:T;\eta[x\mapsto m]} \\
[\![e_1\; e_2]\!]_{\Gamma;\eta} & = & [\![e_1]\!]_{\Gamma;\eta} \cdot [\![e_2]\!]_{\Gamma;\eta} \\
[\![\mathit{put}[S,T]]\!]_{\Gamma;\eta} & = & \mathrm{Put}_{\Gamma;\eta}[S,T] \\
[\![\mathit{fun}\,(X{\leq}U)\; e]\!]_{\Gamma;\eta} & = & \lambda p.\; [\![e]\!]_{\Gamma,X\leq U;\eta[X\mapsto p]} \\
[\![e\,T]\!]_{\Gamma;\eta} & = & [\![e]\!]_{\Gamma;\eta} \cdot \mathrm{Put}_{\Gamma;\eta}[T, \mathrm{Bound}_\Gamma(e)]
\end{array}$$

4.3.13 Lemma: If $[\![e_1]\!]_{\Gamma,\,x:S;\eta[x\mapsto[\![e_2]\!]_{\Gamma;\eta}]}$ is defined, then $[\![[e_2/x]e_1]\!]_{\Gamma;\eta}$ is also defined and the two expressions are equal.

Proof: By induction on the structure of e_1. In the cases for *put* and type abstraction and application, we make use of the fact that term variable assumptions do not affect the interpretation of subtyping (4.3.6). □

4.3.14 Lemma [Stability of type substitutions in terms]: Suppose

$$\begin{array}{l}
\Gamma,\, X{\leq}U,\, \Delta \vdash e \in T \\
\Gamma \vdash S \leq U \\
(\eta,\, \tau) \models \Gamma,\, X{\leq}U,\, \Delta \\
\tau(X) = [\![S]\!]_\tau \\
\eta(X) = \mathrm{Put}_{\Gamma;\eta}[S,U].
\end{array}$$

Then:

$$[\![e]\!]_{\Gamma,\,X\leq U,\,\Delta;\eta}\ \{[\![T]\!]_\tau\}\ [\![[S/X]e]\!]_{\Gamma,\,[S/X]\Delta;\eta}$$

Proof: By induction on the structure of a derivation of $\Gamma,\, X{\leq}U,\, \Delta \vdash e \in T$. The case where the final rule in this derivation is T-Sub follows from the soundness of subtyping, Definition 4.2.2, and the induction hypothesis. The cases for abstraction, application, pairing, projection, and term variables are straightforward. For type abstraction, we use the induction hypothesis with Δ extended by the abstracted variable. The case for T-Put is an instance of Lemma 4.3.11. Finally, for T-All-E, we must have

$$\begin{array}{l}
e = e'\ P \\
\Gamma,\, X{\leq}U,\, \Delta \vdash e' \in \mathit{All}\,(Y{\leq}V)\, T' \\
\Gamma,\, X{\leq}U,\, \Delta \vdash P \leq V \\
T = [P/Y]T'.
\end{array}$$

We then calculate as follows:

$$\begin{array}{cll}
 & [\![e]\!]_{\Gamma,\,X\leq U,\,\Delta;\eta} & \\
= & [\![e']\!]_{\Gamma,\,X\leq U,\,\Delta;\eta} \cdot \mathrm{Put}_{\Gamma,\,X\leq U,\,\Delta;\eta}[P,V] & \text{by definition} \\
\sim & [\![[S/X]e']\!]_{\Gamma,\,[S/X]\Delta;\eta} \cdot \mathrm{Put}_{\Gamma,\,[S/X]\Delta;\eta}[[S/X]P, [S/X]V] & \text{by IH and 4.3.11} \\
= & [\![[S/X]e]\!]_{\Gamma,\,[S/X]\Delta;\eta} &
\end{array}$$

as desired. □

4.3.15 Lemma: Let η_1, η_2 be term environments and let τ be a type environment such that (η_1, τ) and (η_2, τ) each satisfy Γ and η_1 and η_2 agree on type variables. If $\eta_1(x)\ \{[\![\Gamma(x)]\!]_{\tau_1}\}\ \eta_2(x)$ for each $x \in \mathrm{FV}(\Gamma)$, then $[\![e]\!]_{\Gamma;\eta_1}\ \{[\![T]\!]_\tau\}\ [\![e]\!]_{\Gamma;\eta_2}$ whenever $\Gamma \vdash e \in T$.

Proof: By induction over a derivation of $\Gamma \vdash e \in T$. The cases for variables, abstractions, applications, pairing, projections, and subsumption follow the standard pattern for per semantics of $F_\leq$ [Bruce and Longo, 1990, Breazu-Tannen *et al.*, 1991]. More interesting are the cases for type abstraction, type application, and *put*; we consider these in order.

Suppose $e = \mathit{fun}\,(X{\leq}U)\,e'$ and $T = \mathit{All}\,(X{\leq}U)\,T'$, with $\Gamma,\, X{\leq}U \vdash e' \in T'$. We must show that $[\![e]\!]_{\Gamma;\eta_1}$ is related to $[\![e]\!]_{\Gamma;\eta_2}$ in $[\![\mathit{All}\,(X{\leq}U)\,T']\!]_\tau$. So let $R \in \mathrm{PER}$ and $p \models R \leq [\![U]\!]_\tau$. We want to prove $[\![e]\!]_{\Gamma;\eta_1} \cdot p\ \{[\![T']\!]_{\tau[X\mapsto R]}\}\ [\![e]\!]_{\Gamma;\eta_2} \cdot p$. Now, expanding the semantic clause for type abstraction, this is equivalent to $[\![e']\!]_{\Gamma,\,X\leq U;\eta_1[X\mapsto p]}\ \{[\![T']\!]_{\tau[X\mapsto R]}\}\ [\![e']\!]_{\Gamma,\,X\leq U;\eta_2[X\mapsto p]}$. Finish by using the induction hypothesis on e', replacing η_i by $\eta_i[X \mapsto p]$ and τ by $\tau[X \mapsto R]$.

Next, suppose $e = e'\ S$ and $T = [S/X]T'$, with $\Gamma \vdash e' \in \mathit{All}\,(X{\leq}U)\,T'$ and $\Gamma \vdash S \leq U$. We must show that $[\![e]\!]_{\Gamma;\eta_1}$ is related to $[\![e]\!]_{\Gamma;\eta_2}$ in $[\![[S/X]T']\!]_\tau$. Using Lemma 4.3.5 and expanding the semantic clause for type application, this becomes

$$[\![e']\!]_{\Gamma;\eta_1} \cdot \mathrm{Put}_{\Gamma;\eta_1}[S, \mathrm{Bound}_\Gamma(e')]\ \{[\![T']\!]_{\tau[X\mapsto[\![S]\!]_\tau]}\}\ [\![e']\!]_{\Gamma;\eta_2} \cdot \mathrm{Put}_{\Gamma;\eta_2}[S, \mathrm{Bound}_\Gamma(e')].$$

By Lemma 4.3.12, we have $\mathrm{Bound}_\Gamma(e') = U$. Thus, from the assumption $\Gamma \vdash S \leq U$ and Theorem 4.3.8, we know that $\mathrm{Put}_{\Gamma;\eta_1}[S, \mathrm{Bound}_\Gamma(e')] \models [\![S]\!]_\tau \leq [\![U]\!]_\tau$. The induction hypothesis yields $[\![e']\!]_{\Gamma;\eta_1}\ \{[\![\mathit{All}\,(X{\leq}U)\,T']\!]_\tau\}\ [\![e']\!]_{\Gamma;\eta_2}$. Expanding the semantic clause for the quantified type and instantiating R by $[\![S]\!]_\tau$ and p by $\mathrm{Put}_{\Gamma;\eta_1}[S, \mathrm{Bound}_\Gamma(e')]$ gives the desired result (using Fact 4.3.6).

Finally, suppose $e = \mathit{put}[S_1, S_2]$ and $T = S_1 \rightarrow S_2 \rightarrow S_1$. Then $[\![e]\!]_{\Gamma;\eta_1} = [\![e]\!]_{\Gamma;\eta_2} = \mathrm{Put}_{\Gamma;\eta_1}[S_1, S_2]$ by Fact 4.3.6. The desired result is then a consequence of Theorem 4.3.8. □

4.3.16 Corollary [Soundness of typing]: Let $(\eta, \tau) \models \Gamma$ and $\Gamma \vdash e \in T$. Then $[\![e]\!]_{\Gamma;\eta} \in \mathrm{dom}\,[\![T]\!]_\tau$.

Proof: Take $\eta_1 = \eta_2 = \eta$ in 4.3.15. □

4.3.17 Theorem [Equational soundness]: Suppose $(\eta,\, \tau) \models \Gamma$ and $\Gamma \vdash e = e' \in T$. Then $[\![e]\!]_{\Gamma;\eta}\ \{[\![T]\!]_\tau\}\ [\![e']\!]_{\Gamma;\eta}$.

Proof: By induction on a derivation of equality. We consider the various cases of Definition 3.2.1 in order.

- Reflexivity follows from 4.3.16, symmetry and transitivity from the fact that $[\![T]\!]_\tau$ is a per.
- For the reduction rules, first suppose that the final rule in the derivation is R-Beta. We must show that

  $$[\![(\mathit{fun}\,(x{:}S)\ e_1)e_2]\!]_{\Gamma;\eta}\ \{[\![T]\!]_\tau\}\ [\![[e_2/x]e_1]\!]_{\Gamma;\eta}. \tag{1}$$

  It is easy to check, using a simple induction on the structure of a derivation of $\Gamma \vdash (\mathit{fun}\,(x{:}S)\ e_1)e_2 \in T$, that $\Gamma \vdash e_2 \in S$ and $\Gamma,\, x{:}S \vdash e_1 \in T$. By the soundness of typing (4.3.16) and Lemma 4.3.13, (1) is equivalent to

  $$[\![(\mathit{fun}\,(x{:}S)\ e_1)e_2]\!]_{\Gamma;\eta}\ \{[\![T]\!]_\tau\}\ [\![e_1]\!]_{\Gamma,\,x:S;\eta[x\mapsto[\![e_2]\!]_{\Gamma;\eta}]}.$$

It follows immediately by expanding the definition that the two sides are equal as codes; thus, it remains only to show that they are in the domain of $[\![T]\!]_\tau$. This again follows from the soundness of typing.

The argument for R-BETA2 is similar, using Lemma 4.3.14.

The remaining reduction rules are actual equalities of codes, as can be seen from the definition of the interpretation of terms.

- The interesting congruence rules are E-ARROW-I, E-ALL-I, and E-ALL-E. Begin by considering E-ARROW-I. Suppose that $\Gamma,\, x{:}S \vdash e = e' \in T$. We must show that

$$[\![\mathit{fun}\,(x{:}S)\,e]\!]_{\Gamma;\eta}\ \{[\![S{\to}T]\!]_\tau\}\ [\![\mathit{fun}\,(x{:}S)\,e']\!]_{\Gamma;\eta}. \tag{2}$$

  By Fact 3.2.2 and the soundness of typing (4.3.16), both $[\![\mathit{fun}\,(x{:}S)\,e]\!]_{\Gamma;\eta}$ and $[\![\mathit{fun}\,(x{:}S)\,e']\!]_{\Gamma;\eta}$ lie in $\mathrm{dom}([\![S{\to}T]\!]_\tau)$; by Fact 4.1.1, (2) will then follow from

$$\forall s{\in}\,\mathrm{dom}([\![S]\!]_\tau).\ [\![e]\!]_{\Gamma,\,x:S;\eta[x\mapsto s]}\ \{[\![T]\!]_\tau\}\ [\![e']\!]_{\Gamma,\,x:S;\eta[x\mapsto s]}.$$

  But for any $s{\in}\,\mathrm{dom}([\![S]\!]_\tau)$, we have immediately that $(\eta[x \mapsto s],\, \tau) \models \Gamma,\, x{:}S$, from which the induction hypothesis yields the desired result.

  Next, consider E-ALL-I. Suppose that $\Gamma,\, X{\leq}S_1 \vdash e = e' \in S_2$. We must show that

$$[\![\mathit{fun}\,(X{\leq}S_1)\,e]\!]_{\Gamma;\eta}\ \{[\![\mathit{All}\,(X{\leq}S_1)\,S_2]\!]_\tau\}\ [\![\mathit{fun}\,(X{\leq}S_1)\,e']\!]_{\Gamma;\eta}.$$

  After expanding definitions and performing a semantic β-reduction, this amounts to showing that, for all pers R and codes p such that $p \models R \leq [\![S_1]\!]_\tau$:

$$[\![e]\!]_{\Gamma,\,X\leq S_1;\eta[X\mapsto p]}\ \{[\![S_2]\!]_{\tau[X\mapsto R]}\}\ [\![e']\!]_{\Gamma,\,X\leq S_1;\eta[X\mapsto p]}$$

  This again follows directly from the induction hypothesis.

  For the case of E-ALL-E, suppose that $\Gamma \vdash e = e' \in \mathit{All}\,(X{\leq}S_1)\,S_2$ and $\Gamma \vdash T \leq S_1$. We must show

$$[\![e\ T]\!]_{\Gamma;\eta}\ \{[\![[T/X]S_2]\!]_\tau\}\ [\![e'\ T]\!]_{\Gamma;\eta}. \tag{3}$$

  By Fact 3.2.2 and Lemma 4.3.12, $\mathrm{Bound}_\Gamma(e) = \mathrm{Bound}_\Gamma(e') = S_1$. Using this observation and the substitution lemma (4.3.5) and expanding definitions, (3) becomes

$$[\![e]\!]_{\Gamma;\eta} \cdot \mathrm{Put}_{\Gamma;\eta}[T, S_1]\ \{[\![S_2]\!]_{\tau[X\mapsto[\![T]\!]_\tau]}\}\ [\![e']\!]_{\Gamma;\eta} \cdot \mathrm{Put}_{\Gamma;\eta}[T, S_1],$$

  which follows from the induction hypothesis by the soundness of subtyping (4.3.8).

  The remaining congruence rules are straightforward.

- The η-conversion laws for term and type applications and surjective pairing for products follow directly from the interpretations of arrow, product, and universal types.

- E-SUB is a consequence of the fact that subtyping is intepreted in particular as inclusion of pers.

- E-Top is immediate from the interpretation of *Top* as the maximal per.
- The three put-laws follow from the soundness of subtyping and Definition 4.2.2. □

The soundness of the equational theory yields a theory of *program equivalences* in the usual sense. We take a program to be a closed expression of some distinguished base type B (for example, *Int*) and interpret B by some discrete PER, so that $m \ \{[\![B]\!]\}\ n$ implies $m = n$. If two programs can be proved equal in the equational theory, then their interpretations are related, and hence yield equal results.

4.3.18 Observation: Since, as we observed, typing information affects the validity of certain equations, one may ask whether the system is still parametric, i.e., whether polymorphic functions are given uniformly for all types. Obviously, the answer is no because, as observed in Section 1, in this case the type $All\,(X{\leq}PointR)\,X{\rightarrow}X$ would contain only identity functions, which is not what we want (or have). However, we believe that a refined theory of parametricity can be obtained for our calculus based on a translation into pure system F upon which a quantified type $All(X{\leq}T)\ldots$ would be translated into a relativized quantification $All(X)(X{\rightarrow}T) \rightarrow (X{\rightarrow}T{\rightarrow}X) \rightarrow \ldots$ generalizing the "Penn translation" of $F_{\leq}$ into pure system F [Breazu-Tannen *et al.*, 1991]. Notice also that the above interpretation gives rise to a compositional translation of our calculus into untyped computation, which, however, is not simply given by type erasure.

5 Encoding Records

Cardelli [1992] has shown how record types and values can be encoded in a calculus with just products and *Top* in such a way that the expected subtyping and typing rules can be derived. We give a simple version of this encoding, and show that our extended semantics of subtyping also generates the expected updating functions for records. Of course, records can also be taken as primitive constructs of the language, especially for purposes of implementation; but the restriction to products and *Top* simplifies the theoretical study of the calculus.

We begin by defining *flexible tuples* as follows:

5.1 Definition: For each $n \geq 0$ and types T_1 through T_n, let

$$[T_1, \ldots, T_n] \stackrel{\text{def}}{=} T_1 \times (T_2 \times \ldots (T_n \times Top) \ldots).$$

In particular, $[\,] = Top$. Similarly, for terms e_1 through e_n, let

$$[e_1, \ldots, e_n] \stackrel{\text{def}}{=} \langle e_1, \langle e_2, \langle \ldots \langle e_n, top \rangle \ \ldots, \rangle\rangle\rangle,$$

where *top* is some element of *Top* (for example, $fun\,(x{:}Top)\,x$). The projection $e.n$ is:

$$e\underbrace{.2.2\cdots.2}_{n-1 \text{ times}}.1$$

From this abbreviation, we immediately obtain the following rules for subtyping and typing.

$$\frac{\Gamma \vdash S_i \leq T_i \quad \text{for each } i \leq m}{\Gamma \vdash [S_1, \ldots, S_m, S_{m+1}, \ldots, S_n] \leq [T_1, \ldots, T_m]} \qquad \text{(S-Tuple)}$$

$$\frac{\Gamma \vdash e_i \in T_i \quad \text{for each } i \leq m}{\Gamma \vdash [e_1, \ldots, e_m] \in [T_1, \ldots, T_m]} \qquad \text{(T-Tuple)}$$

$$\frac{e \in [T_1, \ldots, T_m]}{\Gamma \vdash e.i \in T_i} \qquad \text{(T-TProj)}$$

The expected reduction rule and equation for surjective tupling are also valid.

Now, let $\mathcal{L} = \{l_1, l_2, \ldots\}$ be a countable set of labels (with a fixed total ordering given by their numeric subscripts). We define *records* as follows:

5.2 Definition: Let L be a finite subset of $\mathcal{L}$ and let S_i be a type for each $l_i \in L$. Let m be the maximal subscript of any element of L, and

$$\hat{S}_i = \begin{cases} S_i & \text{if } l_i \in L \\ \mathit{Top} & \text{if } l_i \notin L. \end{cases}$$

The record type $\{l_i{:}S_i \mid l_i \in L\}$ is defined as the flexible tuple $[\hat{S}_1, \ \hat{S}_2, \ \ldots, \ \hat{S}_m]$. Similarly, if e_i is a term for each $l_i \in L$, then

$$\hat{e}_i = \begin{cases} e_i & \text{if } l_i \in L \\ \mathit{top} & \text{if } l_i \notin L. \end{cases}$$

The record value $\{l_i{=}e_i \mid l_i \in L\}$ is $[\hat{e}_1, \ldots, \hat{e}_m]$. The projection $e.l_i$ is just the tuple projection $e.i$.

We obtain the standard rules for typing and subtyping

$$\frac{J \subseteq I \qquad \Gamma \vdash S_i \leq T_i \quad \text{for each } l_i \in J}{\Gamma \vdash \{l_i{:}S_i \mid l_i \in I\} \leq \{l_i{:}T_i \mid l_i \in J\}} \qquad \text{(S-Rcd)}$$

$$\frac{\Gamma \vdash e_i \in T_i \quad \text{for each } l_i \in I}{\Gamma \vdash \{l_i{=}e_i \mid l_i \in I\} \in \{l_i{:}T_i \mid l_i \in I\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash e \in \{l_i{:}T_i \mid l_i \in I\}}{\Gamma \vdash e.l_i \in T_i} \qquad \text{(T-RProj)}$$

and the evident extensions of the reduction and equality rules. In the following, we usually revert to the more conventional notation for records: $\{x{=}5, \ c{=}\mathit{blue}\} \in \{x{:}\mathit{Int}, \ c{:}\mathit{Color}\}$.

Since records are simple abbreviations for expressions in our core calculus, all of the theory developed so far can be applied directly. In particular, from S-Rcd we obtain an updating function for records:

$$\mathit{put}[\{x{:}\mathit{Int}, \ c{:}\mathit{Color}\}, \{x{:}\mathit{Int}\}] \ \{x{=}5, \ c{=}\mathit{blue}\} \ \{x{=}7\} \triangleright \{x{=}7, \ c{=}\mathit{blue}\}$$

This *put* function is sometimes called *deep update* in the literature: it can be used to overwrite parts of nested data structures.

$$\begin{array}{l} \mathit{put}[\{a{:}\{x{:}\mathit{Int}, \ y{:}\mathit{Int}\}, \ c{:}\mathit{Color}\}, \{a{:}\{x{:}\mathit{Int}\}\}] \\ \qquad \{a{=}\{x{=}5, \ y{=}5\}, \ c{=}\mathit{blue}\} \\ \qquad \{a{=}\{x{=}7\}\} \\ \triangleright \{a{=}\{x{=}7, \ y{=}5\}, \ c{=}\mathit{blue}\} \end{array}$$

By varying the built-in subtyping rule to make cartesian products non-variant on the left and covariant on the right, we could alternatively obtain width-wise record subtyping and updating.

Moreover, the notion of non-overlapping types from Section 3.4 extends naturally to records. (In fact, this extension motivates the term "non-overlapping.")

5.3 Proposition:

1. Suppose that $S_i \perp T_i$ for each $i \leq \min(m,n)$. Then

$$[S_1, \ldots, S_m] \perp [T_1, \ldots, T_n].$$

2. Let I and J be sets of labels and suppose that $S_i \perp T_i$ for each $l_i \in I \cap J$. Then

$$\{l_i{:}S_i \mid l_i \in I\} \perp \{l_i{:}T_i \mid l_i \in J\}.$$

Proof: (1) is an immediate consequence of the product and *Top* cases of the definition of non-overlapping types. (2) then follows directly from the encoding of records as tuples. □

For example, if $e \in \{x{:}Int,\ c{:}Color\}$, then

$$put[\{x{:}Int,\ c{:}Color\}, \{x{:}Int\}]\ e\ \{x{=}7\} = e \in \{c{:}Color\}$$

by Proposition 3.4.3.

6 Application to Objects and Inheritance

Pierce and Turner [1994] presented a model of objects, message-passing, and inheritance based on the typed λ-calculus $F^{\omega}_{\leq}$ [Cardelli, 1990, Compagnoni, 1994, Pierce and Steffen, 1995], an extension of $F_{\leq}$ with higher-order polymorphism. In this section, we apply our refined subtyping mechanism to show how this model of objects can be simplified and extended; we then use the equational theory of Section 3 to prove some simple properties of a small class hierarchy. Our account of inheritance based on positive subtyping extends the treatment in [Pierce and Turner, 1994] in two ways. First, since updating is built in, we obtain a much simpler inheritance operation which avoids explicit handling of update functions. Second, using the equational theory we can reason about object-oriented programs and structure proofs and specifications along the hierarchy of inheritance.

To keep the discussion simple, we work in the second-order system developed in previous sections; this calculus is expressive enough to illustrate the key points of the example, which concern the *creation* of objects from classes. Ultimately, we need to introduce the higher-order polymorphism of $F^{\omega}_{\leq}$ in order to write programs that *use* objects by sending them messages, but this will not concern us here.

Naraschewski, in collaboration with Hofmann, has used the LEGO proof checker to experiment with mechanical verification of object-oriented programs along similar lines [Naraschewski, 1994]. The relationship of these reasoning techniques to the literature on non-type-theoretic object-oriented verification (see, e.g., [Leavens and Weihl, 1992]) remains unexplored.

6.1 Technical Preliminaries

Before starting, we need a little technical machinery. First, we want a logic in which proofs about programs may be phrased — a many-sorted, first-order extension of the equational theory given in Section 3. In particular, we need a notation for predicates on programs and a framework for showing that a function preserves a predicate. Second, to assign a value to *self* when creating objects, a fixed-point combinator is needed [Cook, 1989, Kamin and Reddy, 1994]. Such combinators come in several forms, with varying degrees of power and technical complexity. We consider these points in order.

6.1.1 A Simple Equational Logic for Positive $F_{\leq}$

We introduce a set of first-order formulas φ including atomic equality formulas and quantification over term and type variables. Sequents in this logic have the form $\Gamma \vdash \Phi \Longrightarrow \psi$, where Φ is a set of formulas and ψ is a formula, all well typed in Γ; we write $\Phi,\, \varphi$ for $\Phi \cup \{\varphi\}$. Derivable sequents are defined by augmenting the equational rules with intuitionistic Gentzen-style rules for the logical connectives.

The set of *raw formulas* is given by the abstract grammar

$$\begin{array}{lll} \varphi & ::= & e_1 = e_2 \in T \;\mid\; \varphi_1 \supset \varphi_2 \;\mid\; \varphi_1 \wedge \varphi_2 \;\mid\; \varphi_1 \vee \varphi_2 \;\mid\; \neg\varphi \;\mid \\ & & \forall x{:}T.\ \varphi \;\mid\; \forall X{\leq}T.\ \varphi \;\mid\; \exists x{:}T.\ \varphi \;\mid\; \exists X{\leq}T.\ \varphi, \end{array}$$

where e and T range over the terms and types of positive $F_{\leq}$. A formula $e_1 = e_2 \in T$ is said to be *well typed* in a context Γ if $\Gamma \vdash e_1 \in T$ and $\Gamma \vdash e_2 \in T$; this notion is extended to arbitrary formulas, taking care of variable binders in the obvious way.

We say that $\varphi(x)$ is a *predicate* on type T in context Γ if φ is a formula well typed in the extended context $\Gamma,\, x{:}T$. We then write $\varphi(e)$ for the substitution instance $[e/x]\varphi$. A *sequent* is an expression $\Gamma \vdash \Phi \Longrightarrow \psi$, where Φ is a finite sets of formulas, each well typed in Γ, and ψ is a single formula, also well typed in Γ.

Suppose $(\eta,\, \tau) \models \Gamma$. An atomic formula $e_1 = e_2 \in T$ (well typed in Γ) is *semantically valid* with respect to $(\eta,\, \tau)$ if $[\![e_1]\!]_{\Gamma;\eta}\ \{[\![T]\!]_{\tau}\}\ [\![e_2]\!]_{\Gamma;\eta}$. Again, semantic validity is extended to arbitrary formulas. A sequent $\Gamma \vdash \{\varphi_1,\, \ldots,\, \varphi_n\} \Longrightarrow \psi$ is *semantically valid* with respect to $(\eta,\, \tau)$ if the formula $(\varphi_1 \wedge \cdots \wedge \varphi_n) \supset \psi$ is semantically valid with respect to $(\eta,\, \tau)$.

A sound and (for present purposes) sufficiently powerful inference system can be obtained by augmenting the usual intuitionistic Gentzen-style rules with suitably generalized "copies" of the equational laws from Section 3. For example, we have logical rules such as

$$\frac{\Gamma \vdash S \leq T \qquad \Gamma \vdash \Phi,\, [S/X]\varphi \Longrightarrow \psi}{\Gamma \vdash \Phi,\, \forall X{\leq}T.\ \varphi \Longrightarrow \psi} \tag{$\forall$-E}$$

$$\frac{\Gamma \vdash \Phi,\, \varphi \Longrightarrow \psi \qquad \Gamma \vdash \Phi \Longrightarrow \varphi}{\Gamma \vdash \Phi \Longrightarrow \psi} \tag{Cut}$$

$$\frac{\Gamma \vdash \Phi \Longrightarrow e_1 = e_2 \in T \qquad \Gamma \vdash \Phi \Longrightarrow [e_1/x]\psi}{\Gamma \vdash \Phi \Longrightarrow [e_2/x]\psi} \tag{Leibniz}$$

and equational rules such as:

$$\frac{\Gamma \vdash \Phi \Longrightarrow e_1 = e_2 \in T}{\Gamma \vdash \Phi \Longrightarrow e_2 = e_1 \in T} \tag{E-Symm}$$

$$\frac{\Gamma,\, x{:}S \vdash \Phi \Longrightarrow e_1 = e_2 \in T \qquad x \text{ not free in } \Phi}{\Gamma \vdash \Phi \Longrightarrow \mathit{fun}\ (x{:}S)\ e_1 = \mathit{fun}\ (x{:}S)\ e_2 \in S{\to}T} \tag{E-Arrow-I}$$

$$\Gamma \vdash \Phi \Longrightarrow \mathit{put}[S,T]\ s\ t = t \in T \tag{E-Put-1}$$

We conjecture that both Cut and Leibniz can be eliminated, but we have not studied the proof-theoretic properties of this system.

The soundness of these rules with respect to semantic validity can be established by mimicking the proof of soundness of the equational theory (4.3.17). Since we have chosen a rather weak set of rules, they should also admit other interpretations, for example realizability interpretations or interpretations with respect to other models of the term language.

In the following, we apply the inference rules informally, implicitly carrying the context Γ and additional premises Φ in the running text.

6.1.2 Fixed Points and Bounded Induction

The unrestricted use of *self* in object-oriented programs can, in principle, give rise to arbitrary patterns of recursion. Such programs can only be interpreted using partial functions and general recursion. We see no obstacle to modifying our semantics so as to accommodate partial functions (using, for example, the complete, uniform pers of Amadio [1991]) and refining our equational theory accordingly, to support inequational reasoning. However, many examples arising from object-oriented practice obey a more restrained discipline in which *self* is used as a program structuring technique rather than as a means of implementing general recursive algorithms. In such cases, the recursion implicit in the use of *self* terminates after a *fixed* number of iterations — typically just one or two, as in the case of the *cpointClass* that we define in Section 6.3. In particular, inheritance with *self* is often used to provide *virtual methods*, like the *display* method in a generic class of geometric objects, which can be called from other methods of the generic class even though their implementations are not supplied by the generic class itself (they are supplied by each concrete subclass). In such situations, the fixed point can be replaced by a *bounded fixed point*, which may be interpreted in a model providing only total functions, such as our existing per semantics. By relying only on the elementary reasoning principles that apply in this setting, we avoid obscuring the salient features of the present case study — the logical reflection of the late binding of methods and the modular structure of proofs following the hierarchy of classes.

For each natural number n, let $\mathit{fix}_n(f,a)$ stand for the expression $f^n(a)$. This abbreviation satisfies the following derived typing rule:

$$\frac{\Gamma \vdash f \in T \to T \qquad \Gamma \vdash a \in T}{\Gamma \vdash \mathit{fix}_n(f,a) \in T} \tag{T-FixN}$$

We say that an instance $\mathit{fix}_n(f,a)$ is *sound* if it is well typed in the prevailing context Γ and if f^n is constant, i.e., we can prove $\Gamma,\ x{:}T,\ y{:}T \vdash f^n(x) = f^n(y) \in T$. Now, if $\mathit{fix}_n(f,a)$ is sound and $\Gamma \vdash b \in T$, then $\mathit{fix}_n(f,b)$ is also sound and $\Gamma \vdash \mathit{fix}_n(f,a) = \mathit{fix}_n(f,b) \in T$. Moreover, $\mathit{fix}_n(f,a)$ is a fixed point of f, i.e., $\Gamma \vdash \mathit{fix}_n(f,a) = f(\mathit{fix}_n(f,a)) \in T$. This gives rise to the following derived rule, which can be seen as a bounded version of the standard fixed-point induction principle:

> Bounded induction: Let $\varphi(x)$ be a predicate on a type T in a context Γ, and let Φ be a set of formulas well typed in Γ. Moreover, suppose $\Gamma \vdash \mathit{fix}_n(f,a) \in T$ is sound. If
>
> 1. φ is *consistent*, i.e., there is some $\Gamma \vdash t \in T$ such that $\Gamma \vdash \Phi \Longrightarrow \varphi(t)$; and
> 2. φ is preserved by f, i.e., $\Gamma \vdash \Phi \Longrightarrow \forall x{:}T.\ \varphi(x) \supset \varphi(f(x))$;
>
> then $\Gamma \vdash \Phi \Longrightarrow \varphi(\mathit{fix}_n(f,t))$.

6.2 Basic Classes

We begin our discussion of objects and classes with a simplified variant, omitting the pseudo-variable *self*. We extend this account in Section 6.3 to include *self*; in Section 6.4 we use our equational logic to verify some properties of a tiny class hierarchy.

An *interface signature* is a type $M(X)$ with a distinguished free variable X. For example, here is the signature of the simple point objects described in the introduction:

$$\begin{aligned} \mathit{PointM}(X) = \{&\mathit{get}\colon X \to \mathit{Int},\\ &\mathit{set}\colon X \to \mathit{Int} \to X,\\ &\mathit{bump}\colon X \to X\} \end{aligned}$$

The idea is that, given a concrete representation type R, an initial state $s \in R$, and a record of methods $m \in M(R)$, we can build a value $\mathit{object}_M(R, s, m)$. The type $\mathit{Object}(M)$ contains all objects constructed in this way; that is, the variable X in M is a place-holder for an arbitrary representation type. A typical implementation of points might use the representation type

$$\mathit{PointR} = \{x{:}\mathit{Int}\}$$

and these methods:

$$\begin{aligned} m = \{&\mathit{get} = \mathit{fun}\,(s{:}\mathit{PointR})\; s.x,\\ &\mathit{set} = \mathit{fun}\,(s{:}\mathit{PointR})\,\mathit{fun}\,(i{:}\mathit{Int})\;\{x = i\},\\ &\mathit{bump} = \mathit{fun}\,(s{:}\mathit{PointR})\;\{x = 1 + s.x\}\} \end{aligned}$$

An object with this representation and collection of methods would then be built by applying *object* to m and an appropriate initial state:

$$p = \mathit{object}_{\mathit{PointM}}(\mathit{PointR},\ \{x = 0\},\ m).$$

Object and *object* can be encoded in pure $F_\leq$ using existential types [Pierce and Turner, 1994], but we are more interested here in convenient ways of constructing the record of methods m. (In the positive subtype relation, *Object* turns out to be an *invariant* type constructor, rather than covariant as is usually expected; we return to this point in Section 7.)

A *class* is a data structure that can be used in two ways: it can be *instantiated* to build the methods of new objects, and it can be extended to form *subclasses* sharing some of its behavior. A class whose instances have interface M contains code for the methods described by M. Moreover, since a class may be reused in a subclass with a different representation type S — typically a record type representing a bigger set of instance variables — the methods in a class must be defined polymorphically with respect to possible extensions of the representation type. Classes thus have the following type:

$$\mathit{Class}(M, R) = \mathit{All}\,(S{\leq}R)\; M(S).$$

For example, for the signature *PointM* and the representation type *PointR*, we may define

$$\begin{aligned} &\mathit{simplePointClass} =\\ &\quad \mathit{fun}\,(S{\leq}\mathit{PointR})\\ &\qquad \{\mathit{get} = \mathit{fun}\,(s{:}S)\; s.x,\\ &\qquad\; \mathit{set} = \mathit{fun}\,(s{:}S)\,\mathit{fun}\,(i{:}\mathit{Int})\; \mathit{put}[S, \mathit{PointR}]\ s\ \{x = i\},\\ &\qquad\; \mathit{bump} = \mathit{fun}\,(s{:}S)\; \mathit{put}[S, \mathit{PointR}]\ s\ \{x = 1 + s.x\}\}\\ &\quad \in \mathit{Class}(\mathit{PointM},\ \mathit{PointR}). \end{aligned}$$

This class can be instantiated as follows, yielding a point with the same behavior as p:

$$myPoint = object_{PointM}(PointR,\ \{x = 0\},\ simplePointClass\ PointR)$$

This presentation of classes is significantly cleaner than the one in [Pierce and Turner, 1994], which required that the coercion and update functions connecting R and S be managed explicitly by the programmer. More importantly, we can use the equational theory of Section 3 to reason about such classes — something that is not just inconvenient but impossible in low-level models like [Pierce and Turner, 1994], where the connecting functions are carried by ordinary variables with no special properties.

Let us show, for example, that, whenever $S \leq PointR$ and $s \in S$ and $i \in Int$,

$$s{:}PointR,\ i{:}Int \vdash (simplePointClass\ S).get\ ((simplePointClass\ S).set\ s\ i) = i \in Int. \tag{4}$$

Proof: Expanding the definition, the left-hand side becomes

$$(put[S, PointR]\ s\ \{x = i\}).x.$$

By E-PUT-1, this equals $\{x = i\}.x$, which equals i by the derived record law E-RCD-PROJ. □

Continuing with the example, we also want to implement colored points, which have a method for querying color information. Their interface signature and intended implementation type are:

$$\begin{array}{lcl} CPointM(X) & = & \{get{:}\ X \to Int, \\ & & \ set{:}\ X \to Int \to X, \\ & & \ getC{:}\ X \to Color, \\ & & \ bump{:}\ X \to X\} \\ CPointR & = & \{x{:}Int,\ c{:}Color\}. \end{array}$$

The crux of the example is that we can define a class of colored points by *inheriting* the behavior of the *get*, *set*, and *bump* methods from *simplePointClass*.

$$\begin{array}{l} simpleCPointClass = \\ \quad fun\ (S \leq CPointR) \\ \quad\quad let\ super\ = simplePointClass\ S \\ \quad\quad in\ \{get = super.get, \\ \quad\quad\quad\ \ set = super.set, \\ \quad\quad\quad\ \ bump = super.bump, \\ \quad\quad\quad\ \ getC = fun\ (s{:}S)\ s.c\} \end{array}$$

The well-typedness of the crucial subexpression *simplePointClass S* follows from transitivity.

Now, suppose that *simplePointClass* satisfies equation (4). Without looking back at the definition of *simplePointClass*, we want to show that *simpleCPointClass* satisfies the analogous equation

$$s{:}PointR,\ i{:}Int \vdash (simpleCPointClass\ S).get\ (simpleCPointClass\ S).set\ s\ i = i \in Int \tag{5}$$

for all $S \leq CPointR$ and $s \in S$ and $i \in Int$.

Proof: Expanding the definition of *simpleCPointClass*, this becomes

$$super.get\ super.set\ s\ i = i,$$

where $super = PointClass\ S$. But this is just an instance of (4). □

6.3 Classes with self

Most object-oriented languages not only allow the implementor of a class to refer to the methods of its superclass, but also provide, via a pseudo-variable *self*, recursive access to the methods of the subclass from which a running object has actually been instantiated. In the theoretical literature on object-oriented languages, this feature has been modeled by abstracting the methods of classes on a variable *self*, which is supplied at instantiation time using a fixed-point operator [Cook, 1989, etc.]. In the present framework, this extension is accomplished by altering the type of classes as follows:

$$Class(M, R) = All\,(S{\le}R)\; M(S) \to M(S).$$

The constructors *Object* and *object* need not be changed. To instantiate a class $cl \in Class(M, R)$, we first take a (bounded) fixed point of $cl(R)$ and then proceed as before:

$$o = object_{M}(R,\, r,\, fix_n(cl(R)))$$

Note that the fixed point of $cl(R)$ is used only to build the methods of the final object; the state $r \in R$ is supplied separately, as always. Also, the fixed point is taken only at object instantiation time, not when subclasses are built, so that "late binding" of methods behaves as expected (e.g., in the definition of *cpointClass* below).

Using *self*, we can rewrite our earlier class of points so that the *bump* method is implemented in terms of the *get* and *set* methods instead of directly modifying the state:

$$\begin{array}{l}
pointClass = \\
\quad fun\,(S{\le}PointR)\;\; fun\,(self{:}PointM(S)) \\
\qquad \{get = fun\,(s{:}S)\; s.x, \\
\qquad\; set = fun\,(s{:}S)\, fun\,(i{:}Int)\; put[S, PointR]\;\, s\; \{x = i\}, \\
\qquad\; bump = fun\,(s{:}S)\; self.set\;\, s\;\, ((self.get\;\, s) + 1)\}
\end{array}$$

Now, for the sake of the example, suppose we want *cpointClass* to override the *set* method from *pointClass* with a new method that invokes the *set* method from *super* and then sets the *c* field of the result to *blue*; that is, the *set* method of *cpointClass* has the same behavior as that of *pointClass* as far as the fields in *PointR* are concerned, but also sets the *c* field. Since the *bump* of *cpointClass* is the same as that of *pointClass*, which is implemented using *set*, the new behavior of *set* will be shared by *bump*: sending *bump* to an instance of *cpointClass* will change its color to *blue*.

$$\begin{array}{l}
cpointClass = \\
\quad fun\,(S{\le}CPointR)\;\; fun\,(self{:}\; CPointM(S)\,) \\
\qquad let\; super\;\; = pointClass\;\, S\;\, self \\
\qquad in\; \{get = super.get, \\
\qquad\qquad set = fun\,(s{:}S)\, fun\,(i{:}Int)\;\; put[S, \{c{:}Color\}]\;\, (super.set\;\, s\;\, i)\;\, \{c = blue\}, \\
\qquad\qquad bump = super.bump, \\
\qquad\qquad getC = fun\,(s{:}S)\; s.c\}
\end{array}$$

The implementations of *set* and *bump* in this tiny class hierarchy already illustrate many of the key features of Smalltalk-style inheritance (including the characteristic intricacy that accompanies the use of *self*).

The process by which subclasses are built from superclasses might be described more abstractly as follows. Suppose M and M' are the interfaces of an existing superclass and a desired subclass, and R and R' are the representation types, with $R' \le R$ and $S{\le}R' \vdash M'(S) \le M(S)$. Let

$superClass \in Class(M, R)$ be the superclass, and let $build \in All\,(S{\leq}R')\;\; M'(S) \rightarrow M(S) \rightarrow M'(S)$ be an extension function that, given $self \in M'(S)$ and $super \in M(S)$, constructs a record of subclass methods specialized for the representation type S. The new subclass is then given by the expression

$$\begin{array}{l} subClass = \\ \quad fun\,(S{\leq}R')\;\; fun\,(self{:}M'(S)\,) \\ \quad\;\; let\ super\ = superClass\ S\ self \\ \quad\;\; in\ build\ S\ self\ \ super. \end{array}$$

6.4 Reasoning about Classes with self

The construction of subclasses from superclasses has an exact analog at the logical level. For each $S \leq R$, let $\varphi_S(m)$ be a predicate on type $M(S)$ — i.e., $\varphi_S(m)$ should be well typed in the context $S{\leq}R$, $m{:}M(S)$. Furthermore, for each $S \leq R'$, let $\varphi'_S(m)$ be a predicate on type $M'(S)$ such that $S{\leq}R',\ m{:}M'(S) \vdash \varphi'_S(m) \Longrightarrow \varphi_S(m)$. Then in order to show that φ'_S is an invariant of the *subClass* defined above — i.e., that $S{\leq}R',\ self{:}M'(S) \vdash \varphi'_S(self) \Longrightarrow \varphi'_S(subClass\ S\ self)$ — it suffices to show that φ_S is an invariant of*superClass* in the same sense and that, furthermore,

$$S{\leq}R',\ self{:}M'(S),\ super{:}M(S) \vdash \varphi'_S(self) \wedge \varphi_S(super) \Longrightarrow \varphi'_S(build\ S\ self\ \ super).$$

This pattern of reasoning is useful because it allows us to prove the crucial premise of the bounded induction principle — stability of a predicate — in a modular fashion, following the hierarchy of classes.

For example, suppose we build a record of colored point methods by taking a fixed point of the methods in *cpointClass*:

$$\begin{array}{rcl} cpointMeth & = & fix_2(cpointClass\ \ CPointR,\ m) \\ & \in & CPointM(CPointR) \end{array}$$

(Here m can be any term of type $CPointM(CPointR)$, for example the simple implementation $simpleCPointClass(CPointR)$. In a more general setting with partial functions, the bottom element would play the role of m.) Let us now prove the following fact about *cpointMeth*:

$$\vdash \forall s{\in}CPointR.\ cpointMeth.get\ \ (cpointMeth.bump\ \ s) = (cpointMeth.get\ \ s) + 1 \in Int \tag{6}$$

Proof: For each $S \leq PointR$, let φ_S be the predicate

$$\begin{array}{rlll} \varphi_S(m) \stackrel{\text{def}}{=} & & \forall s{\in}S.\ m.get\ \ (m.bump\ \ s) = (m.get\ \ s) + 1 & \in Int \\ & \wedge & \forall s{\in}S.\ m.get\ \ s = s.x & \in Int \\ & \wedge & \forall s{\in}S.\ \forall i{\in}Int.\ m.set\ \ s\ \ i = \{x{=}i\} & \in PointR \end{array}$$

of type $PointM(S)$ (in the empty context). As usual in inductive proofs, this predicate is somewhat stronger than the property we actually wish to prove. The two auxiliary clauses can be thought of as specifying the salient aspects of the behavior of the recursively invoked methods *get* and *set*. This φ plays the roles of both φ and φ' above.

We first use straightforward equational and logical reasoning to show that φ_S is preserved by *pointClass*:

$$S{\leq}PointR,\ self{:}PointM(S) \vdash \varphi_S(self) \Longrightarrow \varphi_S(pointClass\ \ S\ \ (self)).$$

We must next show that φ_S is preserved by *cpointClass* S for each $S \leq \mathit{CPointR}$. So suppose that $\mathit{self} \in \mathit{CPointM}(S)$ satisfies φ_S. From the fact that φ_S is preserved by *pointClass* S, we have that φ_S holds for

$$\mathit{super} = \mathit{pointClass}\ S\ \mathit{self},$$

which immediately gives the first two clauses in $\varphi_S(\mathit{cpointClass}\ S \mathit{self})$, since the *get* and *bump* fields of *cpointClass* S *self* are copied directly from *super*. After expanding the definition of *set* in *cpointClass*, the third clause becomes

$$\forall s{\in}S.\ \forall i{\in}\mathit{Int}.\ \mathit{put}[S, \{c{:}\mathit{Color}\}]\ (\mathit{super.set}\ s\ i)\ \{c = \mathit{blue}\} = \{x{=}i\} \in \mathit{PointR}. \tag{7}$$

Now, since $\mathit{PointR} = \{x{:}\mathit{Int}\}$ does not overlap with $\{c{:}\mathit{Color}\}$, the left-hand side equals $\mathit{super.set}\ s\ i$ in *PointR*, by Proposition 3.4.3. Equation (7) follows from the third clause of $\varphi_S(\mathit{super})$.

To apply bounded induction, we must still show that the instance of *fix* in the definition of *cpointMeth* is sound, and that $\varphi_{\mathit{CPointR}}$ is consistent.

Soundness is easily checked by examining the tree of possible calls through *self*. It is worth pointing out that this step requires some *global* — i.e. non-modular — inspection of the code of the two classes. For reasoning about larger programs, we might prefer to impose some syntactic constraint that could be checked locally for each class definition. (E.g., we might place a well-founded ordering on the set of methods and only allow the use of *self* if the method invoking *self* falls higher in the ordering than the method being invoked through *self*. The subscript of *fix* could then be generated automatically.)

The consistency of $\varphi_{\mathit{CPointR}}$ is witnessed by the simple implementation *simplePointClass*(*CPointR*). Bounded induction now yields $\varphi_{\mathit{CPointR}}(\mathit{cpointMeth})$. Equation (6) is the first clause of this formula. □

A different reasoning paradigm underlies the proof of the following fact, which highlights the "late binding" of methods invoked through *self*:

$$\vdash \forall s{\in}\mathit{CPointR}.\ \mathit{cpointMeth.getC}\ (\mathit{cpointMeth.bump}\ s) = \mathit{blue} \in \mathit{Color}. \tag{8}$$

Proof: We must first isolate some property that describes the way *bump* calls *set* in *pointClass* (and thus *cpointClass*). Let S be a subtype of *PointR*. Expanding the definition of *pointClass*, we can prove

$$\forall \mathit{self}{\in}\mathit{PointM}(S).\ \forall s{\in}S.\ \exists i{\in}\mathit{Int}.\ (\mathit{pointClass}\ S\ \mathit{self}).\mathit{bump}\ s = \mathit{self.set}\ s\ i \in S, \tag{9}$$

that is, the *bump* method modifies the state by making exactly one call to *set*.

For any $S \leq \mathit{CPointR}$ and $m \in \mathit{CPointM}(S)$, let

$$\begin{array}{lll} \varphi_S(m) \stackrel{\text{def}}{=} & \forall s{\in}S.\ m.\mathit{getC}\ (m.\mathit{bump}\ s) = \mathit{blue} & \in \mathit{Color} \\ & \wedge\ \ \forall s{\in}S.\ \forall i{\in}\mathit{Int}.\ m.\mathit{getC}\ (m.\mathit{set}\ s\ i) = \mathit{blue} & \in \mathit{Color}. \end{array}$$

Our aim is to prove $\varphi_{\mathit{CPointR}}(\mathit{cpointMeth})$ by bounded induction. Begin by observing that the simple implementation *simpleCPointClass* S may easily be modified to show consistency of φ_S. Next, suppose that $\mathit{self} \in \mathit{CPointM}(S)$ satisfies φ_S. We must show $\varphi_S(\mathit{cpointClass}\ S\ \mathit{self})$. The second clause is immediate from the definition of *cpointClass* using E-Put-1 and R-ProjR. On the other hand, after expanding definitions the first clause becomes

$$\forall s{\in}S.\ ((\mathit{pointClass}\ S\ \mathit{self}).\mathit{bump}\ s).c = \mathit{blue} \in \mathit{Color}.$$

Using (9), we can rewrite the right-hand side as $(\mathit{self.set}\ s\ i).c$ for some i. This, in turn, equals *blue* by the second clause of $\varphi_S(\mathit{self})$. This completes the argument. □

7 Extensions

In full $F_{\leq}$, where arrows are contravariant in their domains, the rule S-PROD is derivable from the standard impredicative encoding of products:

$$S_1 \times S_2 \stackrel{\text{def}}{=} All\,(X \leq Top)\;(S_1 \to S_2 \to X) \to X.$$

In positive $F_{\leq}$, this encoding fails to satisfy S-PROD, which is why we included cartesian product as a primitive type constructor.

Similarly, the impredicative encoding of existential types,

$$Some(X \leq S_1)S_2 \stackrel{\text{def}}{=} All\,(Y \leq Top)\;\;(All\,(X \leq S_1)\;S_2 \to Y) \to Y$$

is non-variant in both S_1 and S_2. However, even if existential types are added as primitives, the rule for comparing two existentials cannot be covariant in their bodies,

$$\frac{\Gamma,\, X \leq U \vdash S \leq T}{\Gamma \vdash Some\,(X \leq U)\;S \leq Some\,(X \leq U)\;T} \qquad \text{(S-SOME)}$$

since this would imply the existence of an updating function $put[Some\,(X \leq U)\;S, Some\,(X \leq U)\;T] \in (Some\,(X \leq U)\;S) \to (Some\,(X \leq U)\;T) \to (Some\,(X \leq U)\;S)$. When S and T are unequal, this function does not make sense; for suppose we are given $\langle P, s\rangle \in Some\,(X \leq U)\;S$ and $\langle Q, t\rangle \in Some\,(X \leq U)\;T$, where P and Q are the witness types of the two existential values. In order to safely overwrite some components of s with the corresponding components of t, it must be the case that P and Q are identical. But this cannot be guaranteed statically. Similar considerations apply to other type constructors such as disjoint unions, variants, and recursive types such as lists, which must be invariant in the positive subtyping relation as no sensible definition of $put[S,T]$ exists in these cases.

Similarly, object types must be invariant under the positive subtyping relation: we have $Object(M) \leq Object(N)$ only when the interface signatures M and N are identical. When object types are modeled using existential types [Pierce and Turner, 1994, Hofmann and Pierce, 1995], this invariance arises from the considerations above. More generally, it does not seem reasonable to expect *any* encoding of objects to support positive subtyping between object types. Intuitively, we would have to require that $Object(M) = Object(N) \times R$ for some R — that is, the behavior of the "N part" of an M object would have to be completely independent of the behavior of the rest of the methods. For example, sending a "set color" message to a colored point would have to leave the position unchanged. (It is illustrative to try defining an update function for the points and colored points of Section 6.3.)

It seems possible to get along without subsumption for object types, both in the object-oriented examples of Section 6 and in the more general constructions in [Pierce and Turner, 1994, Hofmann and Pierce, 1995], at the cost of replacing some implicit coercions between object types by polymorphic type applications. Suppose we want to define a function f that operates on both points and colored points. Instead of defining f by

$$f = fun\,(p{:}Object(PointM))\;\ldots$$

and applying it to a colored point cp by writing just $f\;cp$, we give f an extra abstraction (on a *type operator* M ranging over interfaces refining the interface of points)

$$f = fun\,(M \leq PointM)\;\;fun\,(p{:}Object(M))\;\ldots$$

and applying f to cp by writing $f\ \ CPointM\ \ cp$.

A more pragmatically satisfying solution, however, would be to enrich the subtype relation so that, instead of providing only positive subtyping, we allow positive and ordinary subtyping to exist side by side, defining the constant $put[S,T]$ only in the case where $S \leq T$ can be proved from just the positive rules. In particular, existential types, sums, lists, and object types would obey the familiar covariant laws in the "full" subtyping relation, but would lack $put[S,T]$ constants. In such a system, positive subtyping would still be used to allow updates to the internal states of objects, while the objects themselves (viewed from outside) would obey the usual principle of subsumption.

Distinguishing two kinds of subtyping slightly complicates the presentation of the system and requires a more careful analysis of the subtyping and typing algorithms, but seems to present no serious difficulties. Martí-Oliet and Meseguer [1990] have also proposed a system with two subtype relations, one containing the other. For them, the "smaller" relation denotes set-theoretic inclusion, which, as in our calculus, requires a restriction of function subtyping to the covariant case.

Another useful extension of positive $F_{\leq}$ (or of full $F_{\leq}$ with a distinguished positive fragment) would allow F^{ω}-style type operators, as in [Cardelli, 1990, Pierce and Steffen, 1995, Compagnoni, 1994]. The subtyping rule for type operators is just the extension of the positive subtype relation on their codomains: if $F(T) \leq G(T)$ for every T, then $F \leq G$. This law should give rise to a polymorphic *put* function $put[F,G] \in All\,(X)\,F(X) \to G(X) \to F(X)$. We expect that our semantics for positive $F_{\leq}$ can be extended straightforwardly to this higher-order calculus using the techniques of [Cardelli and Longo, 1991, Compagnoni and Pierce, 1995]. A further extension would accommodate covariance and functorial strength as in [Hofmann and Pierce, 1995], which would provide generic functions for method invocation and also allow automatic insertion of *put* functions in class definitions.

In this paper, we have assigned meaning to terms by giving a PER model providing both a simple operational semantics and justification of a set of equational rules. For larger calculi (including, for example, general fixed-point operators or recursive types, which are often found in more sophisticated models of objects and inheritance), the construction of PER models becomes much more complex [Abadi and Plotkin, 1990, Amadio, 1991]. In this case, one would instead define an *evaluation relation* directly on the syntax of terms, extending the reduction rules in Section 3.1. The equational theory of this system would be based on contextual equivalence in the sense of [Gordon, 1994, etc.].

Acknowledgements

This research was carried out at the University of Edinburgh's Lab for Foundations of Computer Science. Hofmann was supported by a European Union HCM fellowship. Pierce was supported by a fellowship from the British Science and Engineering Research Council. We have profited from discussions of records and objects with Martín Abadi, Luca Cardelli, Didier Rémy, John Reynolds, and Dilip Sequeira. Adriana Compagnoni, David Turner, and members of the FOOL working group gave us useful feedback on the present system. Kathleen Fisher and three anonymous referees made many suggestions for improvement in a late draft.

References

[Abadi and Cardelli, 1994] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software (TACS), Sendai,* *Japan*, 1994.

[Abadi and Plotkin, 1990] M Abadi and G. D. Plotkin. A PER model of polymorphism and recursive types. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 355–365, 1990.

[Amadio, 1991] Roberto M. Amadio. Recursion over realizability structures. *Information and Computation*, 90(2):55–85, 1991.

[Breazu-Tannen *et al.*, 1991] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Bruce and Longo, 1990] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.

[Bruce, 1994] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. A preliminary version appeared in POPL 1993 under the title "Safe Type Checking in a Statically Typed Object-Oriented Programming Language".

[Cardelli and Longo, 1991] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

[Cardelli and Mitchell, 1991] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994); available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

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

[Cardelli *et al.*, 1994] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. A preliminary version appeared in TACS '91 (Sendai, Japan, pp. 750–770).

[Cardelli, 1984] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[Cardelli, 1990] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[Cardelli, 1992] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. Also in Carl A. Gunter and John C.

Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Compagnoni and Pierce, 1995] Adriana B. Compagnoni and Benjamin C. Pierce. Multiple inheritance via intersection types. *Mathematical Structures in Computer Science*, 1995. To appear. Preliminary version available as University of Edinburgh technical report ECS-LFCS-93-275 and Catholic University Nijmegen computer science technical report 93-18, Aug. 1993.

[Compagnoni, 1994] Adriana B. Compagnoni. Subtyping in $F^{\omega}_{\wedge}$ is decidable. Technical Report ECS-LFCS-94-281, LFCS, University of Edinburgh, January 1994. To appear in the proceedings of Computer Science Logic, September 1994, under the title "Decidability of Higher-Order Subtyping with Intersection Types".

[Cook, 1989] William Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[Curien and Ghelli, 1992] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[de Bruijn, 1972] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Fisher and Mitchell, 1994] Kathleen Fisher and John Mitchell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspects of Computer Software, Sendai, Japan*, pages 844–885. Springer-Verlag, April 1994. LNCS 789.

[Gordon, 1994] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, 1994.

[Gunter, 1992] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* The MIT Press, Cambridge, MA, 1992.

[Hofmann and Pierce, 1995] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 1995. To appear. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[Kamin and Reddy, 1994] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994.

[Leavens and Weihl, 1992] Gary T. Leavens and William E. Weihl. Subtyping, modular specification, and modular verification for applicative object-oriented programs. Technical Report TR#92-28d, Department of Computer Science, Iowa State University, September 1992. Revised September, October 1993, and January, 1994. Portions to appear in *Acta Informatica.*

[Martí-Oliet and Meseguer, 1990] Narciso Martí-Oliet and José Meseguer. Inclusions and subtypes. Technical Report SRI-CSL-90-16, Computer Science Laboratory, SRI International, December 1990.

[Naraschewski, 1994] Wolfgang Naraschewski. *Verifikation objektorientierter Programme mit* LEGO. Studienarbeit, Universität Erlangen, 1994. To appear.

[Oles, 1985] Frank J. Oles. Type algebras, functor categories, and block structure. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*. Cambrige University Press, 1985.

[Pierce and Steffen, 1995] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 1995. To appear. A preliminary version appeared in IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), June 1994, and as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94, January 1994.

[Pierce and Turner, 1994] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Pierce, 1994] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). A preliminary version appeared in POPL '92.

[Reynolds, 1985] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development*. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

[Robinson and Tennent, 1988] Edmund Robinson and Robert Tennent. Bounded quantification and record-update problems. Message to `Types` electronic mail list, October 1988.