Subtyping with Singleton Types

David Aspinall

Department of Computer Science, University of Edinburgh, U.K.
e-mail: `da@dcs.ed.ac.uk`

Abstract. We give syntax and a PER-model semantics for a typed λ-calculus with subtypes and *singleton* types. The calculus may be seen as a minimal calculus of subtyping with a simple form of dependent types. The aim is to study singleton types and to take a canny step towards more complex dependent subtyping systems. Singleton types have applications in the use of type systems for specification and program extraction: given a program P we can form the very tight specification $\{P\}$ which is met uniquely by P. Singletons integrate abbreviational definitions into a type system: the hypothesis $x : \{M\}$ asserts $x = M$. The addition of singleton types is a non-conservative extension of familiar subtyping theories. In our system, more terms are typable and previously typable terms have more (non-dependent) types.

1 Introducing Singletons and Subtyping

Type systems for current programming languages provide only coarse distinctions amongst data values: `Real`, `Bool`, `String`, etc. Constructive type theories for program specification can provide very fine distinctions such as $\{x \in \texttt{Nat} \mid \mathit{Prime}(x)\}$, but often terms contain non-computational parts, or else type-checking is undecidable. We want to study type systems in between where terms do not contain unnecessary codes and, ideally, type-checking is decidable. When types express requirements for data values more accurately, it can help to eliminate more run-time errors and to increase confidence in program transformations which are type-preserving.

Singleton types express the most stringent requirement imaginable. Suppose *fac* stands for the expression:

$$\mu f.\, \lambda x.\, \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (f(x-1))$$

Then $\{\mathit{fac}\}$ is a specification of the factorial function, and

$$\mathit{fac} \ : \ \{\mathit{fac}\}$$

says that *fac* satisfies the specification $\{\mathit{fac}\}$. This is an instance of the principal assertion for singleton types, $M : \{M\}$. But syntactic identity is too stringent; we can write the

This paper appears in *Proc. CSL'94*, Kazimierz, Poland. Springer Lecture Notes in Computer Science, 1995.

factorial function in other ways and it would be useful if when fac' is an implementation of the factorial function, we also have $fac' : \{fac\}$. This suggests that we let $\{M\}$ stand for the collection of terms equal to M in some theory of equality, so $\{M\}$ denotes an equivalence class of terms, rather than a singleton set.

Although we want types to be more expressive, this should not sacrifice the usability of the type system. More types can lead to more polymorphism: a term may possess several types, and the type system should recognize this and allow the programmer as much flexibility as possible. Subtyping systems provide flexibility by allowing a term of some type A to be used where one of a 'larger' type B is expected. The characteristic rule for subtyping is known as *subsumption*, which captures this kind of polymorphism:

$$\frac{M : A \qquad A \leq B}{M : B} \tag{SUB}$$

Subsumption at ground types suggests subtyping at higher types. For example, a function defined on `Int` may be used where one defined only on `Nat` is needed, because $\texttt{Nat} \leq \texttt{Int}$ means that every natural number will be a suitable argument. So we expect that $\texttt{Int} \to \texttt{Int}$ is a subtype of $\texttt{Nat} \to \texttt{Int}$.

Subtyping leads to a stratified notion of equality. Because terms may have many types, the equality of two terms can be different at different types. Indeed, consider two different functions on integers:

$$(\lambda x{:}\texttt{Int}.\ \texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ 2 * x) \;\neq\; (\lambda x{:}\texttt{Int}.\, x) \quad : \texttt{Int} \to \texttt{Int}$$

which have equal values at every natural:

$$(\lambda x{:}\texttt{Int}.\ \texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ 2 * x) \;=\; (\lambda x{:}\texttt{Int}.\, x) \quad : \texttt{Nat} \to \texttt{Int}$$

If in some context only arguments of type `Nat` are supplied, these functions are interchangeable; useful perhaps for program transformation during compilation.

This view of equality influences our treatment of singleton types. Because equality can vary at different types, we think of $\{M\}$ as a family of equivalence classes indexed by a type. We attach a tag to the singleton, which denotes the type at which we "view" the term. The introduction rule for singletons is:

$$\frac{M : A}{M : \{M\}_A} \tag{\{\}-I}$$

In fact, the type tag can be important for another reason: the type A might affect the interpretation of M, as well as its equivalence class of terms (although this is not the case for the semantics we give later). Imagine a model in which the integers are constructed using pairs of naturals: the pair $\langle m, n\rangle$ codes the integer $(m - n)$. Then the interpretation $[\![3{:}\texttt{Int}]\!]$ is quite different from $[\![3{:}\texttt{Nat}]\!]$, and the semantic types have different equality relations associated. (Of course, there is an obvious coercion from $[\![\texttt{Nat}]\!]$ to $[\![\texttt{Int}]\!]$.) To allow for typed interpretations we need to know the type given to a term in a singleton, but unless it is recorded somehow it cannot be determined from a typing derivation.

There is no typing elimination rule for singleton types, but we have a subtyping rule that says that a singleton type is a subtype of its type tag:

$$\frac{M : A}{\{M\}_A \leq A} \qquad (\textsc{sub}\text{-}\{\})$$

which allows us to deduce $M : A$ from $M : \{N\}_A$ via ($\textsc{sub}$).

For us, singleton types have a *non-informative* flavour. In other words, we have no term operators corresponding to singleton introduction and elimination. This contrasts with constructive type theories utilising propositions-as-types, where singletons might be treated akin to a propositional equality type and given a powerful elimination operator. In our approach, membership of singletons corresponds to definitional equality, which is usually decidable. A technical side-effect of non-informative types is that the meta-theory of our system is harder to deal with, because the rules are less syntax-directed.[1] Notice that the presence of ($\textsc{sub}$) already means that the typing rules are not syntax-directed.

The theory of equality we choose to incorporate in singleton types is a natural typed equational theory for the terms. The typing assertion $M : \{N\}_A$ asserts that M and N are equal at type A, so instead of axiomatizing a separate judgement form $\Gamma \vdash M = N : A$, we use *typing* rules with the form $\Gamma \vdash M : \{N\}_A$ directly. The usual rule of β-equality is admissible. This formulation is nicer to deal with than one defined using a rule of untyped β-conversion.

The system $\lambda_{\leq}$ ("lambda-sub")[Car88a] is formed by adding subtyping to the simply-typed λ-calculus. In the remainder of the paper we shall study the addition of singleton types to $\lambda_{\leq}$; we call the resulting system $\lambda_{\leq\{\}}$ ("lambda-sub-singleton"). First, the next section outlines some uses for type systems with singleton types. Then in Section 3 we present the complete definition of $\lambda_{\leq\{\}}$ and in Section 4 we establish some of its meta-theoretic properties. Section 5 gives a PER semantics and shows soundness; Section 6 concludes.

2 Using Singleton Types

Singleton types as specifications. This work arose from a desire to understand the formal system of a specification language called ASL+ [SST92, Asp95]. ASL+ extends the algebraic specification language ASL [SW83] with constructs from type-theory (principally λ-abstraction) for parameterising specifications and programs. ASL consists of a collection of *specification building operators* (SBOs) which are used for putting together specifications; ASL-level specifications form the base types of the ASL+ λ-calculus.

[1] A set of rules is called *syntax-directed* if the last rule used in a derivation of any statement J is uniquely determined by the structure of J.

One of the extensions provided by ASL+ is the ability to express specifications of parameterised programs (like *functor signatures* in Extended ML [KST94]). To do this, we need dependent function spaces to specify a function which satisfies a specification that depends on its argument. Singletons turn a program into a very tight specification which we can use with other SBOs in the body of the function specification. A trivial example is an identity functor, which returns its argument:

$$\texttt{functor Id(}X\texttt{:}S\texttt{):}S\texttt{ = }X$$

In ASL+, this could be specified by $\Pi X{:}\,S.\,\{X\}$. A less trivial example is *SORT*, which specifies a parameterised program that, given a program implementing an order relation *Ord* (a type t and an ordering le on t), returns a sorting function $sort$ for sorting lists of elements of t according to le.

$$\begin{array}{lll} SORT & =_{\mathrm{def}} & \Pi\ Ord{:}ORD\ . \\ & & \quad \textbf{enrich}\ \{Ord\} \\ & & \quad \textbf{by} \\ & & \qquad \textbf{sign}\ sort\ :\ t\ list\ \to\ t\ list \\ & & \qquad \textbf{axioms}\ \text{(some axioms specifying that } sort(x) \text{ is a} \\ & & \qquad\qquad\qquad \text{sorted copy of } x \text{ with respect to } le) \\ & & \quad \textbf{end} \end{array}$$

The important thing to notice here is the use of $\{Ord\}$ to require that applications of programs satisfying *SORT* should be an extension of the actual parameter with a sorting function that operates on exactly the same type t.

Singleton types and subtyping. Adding singleton types to well-known subtyping systems such as $\lambda_{\leq}$ and its second-order variants is not a conservative extension. More typing statements become provable, both because more terms are typable and because terms have more (non-dependent) types.

We illustrate this with a simple example. Consider the identity function on real numbers:

$$id_{\texttt{Real}} \ =_{\mathrm{def}} \ \lambda x{:}\,\texttt{Real}.\,x$$

Suppose the fact that $\texttt{Int} \leq \texttt{Real}$. Then in $\lambda_{\leq}$:

$$\begin{array}{lll} id_{\texttt{Real}} & : & \texttt{Real} \to \texttt{Real} \\ & : & \texttt{Int} \to \texttt{Real} \\ & \not: & \texttt{Int} \to \texttt{Int} \end{array}$$

But the third typing is perfectly reasonable; the identity function on reals certainly maps integers to integers. It is provable in our system, via an extended rule for λ-introduction:

$$\frac{\Gamma \vdash \texttt{Int} \leq \texttt{Real} \qquad \Gamma, x{:}\texttt{Real} \vdash x : \texttt{Real} \qquad \Gamma, x{:}\texttt{Int} \vdash x : \texttt{Int}}{\Gamma \vdash \lambda x{:}\texttt{Real}.\,x : \texttt{Int} \to \texttt{Int}}$$

The second hypothesis ensures that the λ-abstraction can be typed. The third hypothesis allows the body to be given a more refined type based on the assumption that the argument type is more refined, according to the first hypothesis. Although a typing rule like this could be added to $\lambda_{\leq}$, the types $\mathtt{Real} \to \mathtt{Real}$ and $\mathtt{Int} \to \mathtt{Int}$ are incomparable, so this would break the desirable property that every typable term has a minimal type. On the other hand, $\lambda_{\leq\{\}}$ does have the minimal type property, which we show in Section 4. Not surprisingly, the minimal types are singletons.

As well as more types for previously typable terms, it follows from the example that more (singleton-free) terms become typable when we add singleton types. Suppose

$$\mathit{twice}_{\mathtt{Int}\to\mathtt{Int}} \;=_{\mathrm{def}}\; \lambda f\colon \mathtt{Int} \to \mathtt{Int}.\, \lambda x\colon \mathtt{Int}.\, f(f(x)).$$

Then $\mathit{twice}_{\mathtt{Int}\to\mathtt{Int}}(\mathit{id}_{\mathtt{Real}}) : \mathtt{Int} \to \mathtt{Int}$ in $\lambda_{\leq\{\}}$, but is untypable in $\lambda_{\leq}$.

There are more complex cases than these. We can assign a type $\Pi x\colon A.\, \{M\}_B$ to a function $\lambda x\colon A.\, M$ that is as informative as the function definition itself, so we can substitute the result of function applications during type-checking and do some amount of equational reasoning. Similarly, we can substitute the arguments supplied to a function into the body before type checking, which is shown next.

Singleton types as definitions. Simple definition by abbreviation is essential for the practical use of a type system in a programming language or proof assistant. If M is a large expression occuring inside N several times, we may write N as

$$x = M \text{ in } N'$$

where N' is the result of replacing occurrences of M in N with the variable x. If definitions are treated formally, they are introduced as a new concept that extends the type theory causing additional complication. With singleton types we get a form of definition in the system for free, and derive similar rules to those given in [HP91, SP94]. The typed definitions of Severi and Poll [SP94] have the form:

$$x = M\colon A \quad \text{in} \quad N$$

This is similar to a λ-abstraction over a singleton type in $\lambda_{\leq\{\}}$, applied to the trivially appropriate argument:

$$(\lambda x\colon \{M\}_A.\, N)M$$

which in turn can be compared with the usual "trick" for writing definitions in systems such as $\lambda^{\to}$ without using a special mechanism:

$$(\lambda x\colon A.\, N)M$$

Severi and Poll point out three reasons for introducing definitions as a fresh concept:

1. The λ-abstraction $\lambda x{:}\,A.\,N$ might not be permitted in the type-system (if N is a type expression, for example).

2. β-reduction replaces all instances of x in N by M, whereas it is useful to be able to replace instances one-by-one when desired.

3. The information that $x = M$ may lead to a typing for N that otherwise would not be possible.

If definitional bindings must be allowed in more places than λ-abstractions, then it seems necessary to introduce a new concept. For the second point, Severi and Poll introduce a new kind of reduction. A δ-reduction replaces a single instance of x with M. We can similarly introduce a new reduction relation (called Γ-reduction) in $\lambda_{\leq\{\}}$:

$$x \longrightarrow_\Gamma M \qquad \text{if} \qquad \Gamma(x) = \{M\}_A$$

It is indexed by a context Γ which contains typing assumptions for variables. When the type of a variable x is a singleton type $\{M\}_A$, then we consider x as being equal to M (at type A), so we may replace occurences of x by M. This reduction can be extended under λ and Π abstractions if certain care is taken; but we will not consider it further in this paper.

Without adding special constructs for definitions we gain the benefit of the third point, extended typing. Interestingly, because of the rules chosen for singleton types, it is not necessary for the term M to be "revealed" to the function body N to make use of the fact that $x = M$ when typing N. The term $(\lambda x{:}\,A.\,N)M$ has exactly the same types as $(\lambda x{:}\,\{M\}_A.\,N)M$, and moreover the two terms are provably equal in our equational theory.

3 The System $\lambda_{\leq\{\}}$

The system $\lambda_{\leq\{\}}$ is the addition of singleton types to $\lambda_{\leq}$. We do not restrict λ-introduction, so we get dependent product types in place of the usual arrow types. The combination of type-dependency, subtyping, and the non-informative aspect of the singleton constructor make fundamental properties like subject reduction more difficult to establish than usual.

Pre-types A, pre-terms M, and pre-contexts Γ are given by the grammar:

$$\begin{array}{lclclcl} A & ::= & P & \mid & \Pi x{:}\,A.\,B & \mid & \{M\}_A \\ M & ::= & x & \mid & \lambda x{:}\,A.\,M & \mid & MN \\ \Gamma & ::= & \langle\rangle & \mid & \Gamma,\, x : A & & \end{array}$$

There is a set of term variables *Var*, ranged over by x. There are no type variables, so we assume a set *PrimTypes* of primitive (or atomic) types, ranged over by P. Subtyping between primitive types is given by a relation, $\leq_{Prim}$ on *PrimTypes*. Restricting $\leq_{Prim}$ to atomic types ensures that subtyping retains a *structural* character; that is, two types related by the subtype relation will have a similar shape. Free and bound variables are defined as usual. We identify pre-types, pre-terms, and pre-contexts that are alpha-convertible, and reserve $\equiv$ for this syntactic equivalence.

We write $\Gamma \subseteq \Gamma'$ if each $x : A$ in Γ is also in Γ'. Beta-reduction is defined as usual over terms, and extended to types compatibly with the type-constructors. Notice that there is no application at the level of types; convertible types can only differ within corresponding singleton components.

We use the following judgement forms:

- context formation, Γ Context
- type formation, $\Gamma \vdash A$
- typing, $\Gamma \vdash M : A$
- subtyping, $\Gamma \vdash A \leq B$

A judgement is *valid* iff it is derived using the rules of Tables 1 and 2. We use $\Gamma \vdash J$ to range over valid judgements.

We also use an equality judgement as a derived form; $\Gamma \vdash M = N : A$ stands for $\Gamma \vdash M : \{N\}_A$ when we think of it as meaning equality between terms. Equality between types is written $\Gamma \vdash A = B$ which means both $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq A$ are valid.

The four judgements are defined simultaneously, in contrast to non-dependent schemes where the subtype relation can be separated from the typing relation. Below we describe the rules related to singleton types; the other rules shown in the tables are mostly standard.

Singleton Types and Equality. Singleton types are formed by the rule (FORM-$\{\}$) and terms of singleton type are introduced by the equality rules, principally reflexivity (EQ-REFL), which is the singleton introduction rule ($\{\}$-I) shown before under a different guise. Symmetry and transitivity are derived using the subtyping rules shown below. We also have the usual rules for equality of λ-abstractions (EQ-λ) and applications (EQ-APP). The rule (EQ-λ) is more flexible than usual: it allows one to derive equalities between functions by examining only a restricted domain. (This rule is forced when one has untagged singletons and the usual equal domains equality, which was the inspiration for adding it). It leads to the admissibility of a correspondingly stronger typing rule, via (SUB) and (SUB-$\{\}$):

$$\frac{\Gamma \vdash A' \leq A \qquad \Gamma, x\colon A \vdash M : B \qquad \Gamma, x\colon A' \vdash M : B'}{\Gamma \vdash \lambda x\colon A.\, M : \Pi x\colon A'.\, B'}$$

which lets us give a more refined type for a function, given a more refined type for its argument. This was used in the $id_{\texttt{Real}}$ example in Section 2.

One might wonder whether we can have deduction from arbitrary hypotheses of equations between terms, assumed via iterated subscripts in the syntax. For example, it holds that $\Gamma, x\colon \{M\}_{\{N\}_A} \vdash M = N : A$. In fact, we have only a pure theory of equality since this judgement presupposes that $\Gamma \vdash M = N : A$. This is because the rule (ADD-HYP) requires that types in contexts must be well-formed; Proposition 4.1 below establishes this formally.

$$\frac{}{\langle\rangle \ \text{Context}} \qquad (\textsc{empty})$$

$$\frac{\Gamma \vdash A \qquad x \notin dom(\Gamma)}{\Gamma, x\colon A \ \text{Context}} \qquad (\textsc{add-hyp})$$

$$\frac{\Gamma \ \text{Context}}{\Gamma \vdash P} \qquad (\textsc{form-prim})$$

$$\frac{\Gamma, x\colon A \vdash B}{\Gamma \vdash \Pi x\colon A.\, B} \qquad (\textsc{form-}\Pi)$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \{M\}_A} \qquad (\textsc{form-}\{\})$$

$$\frac{\Gamma \ \text{Context}}{\Gamma \vdash x : \Gamma(x)} \qquad (\textsc{var})$$

$$\frac{\Gamma, x\colon A \vdash M : B}{\Gamma \vdash \lambda x\colon A.\, M : \Pi x\colon A.\, B} \qquad (\lambda)$$

$$\frac{\Gamma \vdash M : \Pi x\colon A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]} \qquad (\textsc{app})$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash M : B} \qquad (\textsc{sub})$$

Table 1: Rules for contexts, formation, and typing.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M = M : A} \qquad \text{(EQ-REFL)}$$

$$\frac{\Gamma \vdash A' \leq A \qquad \Gamma, x{:}\,A' \vdash M = M' : B' \qquad \Gamma, x{:}\,A \vdash M : B}{\Gamma \vdash \lambda x{:}\,A.\,M = \lambda x{:}\,A'.\,M' \; : \Pi x{:}\,A'.\,B'} \qquad \text{(EQ-}\lambda\text{)}$$

$$\frac{\Gamma \vdash M = M' : \Pi x{:}\,A.\,B \qquad \Gamma \vdash N = N' : A}{\Gamma \vdash MN = M'N' \; : B[N/x]} \qquad \text{(EQ-APP)}$$

Note: $\Gamma \vdash M = N : A$ is short for $\Gamma \vdash M : \{N\}_A$.

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \leq A} \qquad \text{(SUB-REFL)}$$

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

$$\frac{\Gamma \;\text{Context} \qquad P \leq_{Prim} P'}{\Gamma \vdash P \leq P'} \qquad \text{(SUB-PRIM)}$$

$$\frac{\Gamma \vdash A' \leq A \qquad \Gamma, x{:}\,A' \vdash B \leq B' \qquad \Gamma, x{:}\,A \vdash B}{\Gamma \vdash \Pi x{:}\,A.\,B \leq \Pi x{:}\,A'.\,B'} \qquad \text{(SUB-}\Pi\text{)}$$

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

$$\frac{\Gamma \vdash M = N : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash \{N\}_A \leq \{M\}_B} \qquad \text{(SUB-EQ-SYM)}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \{M\}_A \leq \{M\}_{\{M\}_A}} \qquad \text{(SUB-EQ-ITER)}$$

Table 2: Rules for equality and subtyping.

Subtyping Singletons. Subtyping of singleton types is provided by three rules. First, we have the rule (SUB-$\{\}$) shown earlier, which asserts that a singleton is a subtype of the type it is tagged with.

The rule (SUB-EQ-SYM) combines two principles. The first is monotonicity of equality with respect to subtyping: if two terms are equal at a type, say $M = N : A$, and $A \leq B$, then $M = N : B$ also. We can express this via subtyping of singleton types. Generally, as we pass from subtype to supertype, the equivalence class of any particular term gets larger, so $\{N\}_A \leq \{N\}_B$ and $M = N : B$ via subsumption. The second principle is symmetry of equality, and again we can express the typing rule by a subtyping one (an economy, since we want both). If $M = N : A$ then the equivalence classes of M and N at A must be the same, in particular $\{N\}_A \leq \{M\}_A$. These are combined to get the single rule (SUB-EQ-SYM).

The third subtyping rule for singletons (SUB-EQ-ITER) deals with the case when a singleton type is tagged with another singleton type. Observe that we can repeat the operation of taking singletons in the syntax, forming $\{M\}_A$, $\{M\}_{\{M\}_A}, \ldots$. We shall consider these types as equal, because singleton types are already the smallest non-empty types we are interested in. And because $\{M\}_A$ inherits equality from A, the equality on terms in $\{M\}_A$ and $\{M\}_{\{M\}_A}$ is the same. We have $\{M\}_{\{M\}_A} \leq \{M\}_A$ alread by (SUB-$\{\}$), for the other inclusion we need (SUB-EQ-ITER).

4 Basic meta theory of $\lambda_{\leq\{\}}$

We state some basic properties of $\lambda_{\leq\{\}}$, which lead to proofs of admissibility of rules such as subject β-reduction and β-equality. We then prove the existence of minimal types. Proofs are by straightforward inductions on typing derivations unless stated.

Proposition 4.1 (Contexts and Substitution)

1. Context formation. *Suppose* Γ *Context, where* $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$. *Then*
   (a) $\Gamma \vdash A_i$ *for* $1 \leq i \leq n$.
   (b) *If* $\Gamma \vdash J$, *then* $FV(J) \subseteq \{x_1, \ldots, x_n\}$.
2. Weakening. *If* $\Gamma \vdash J$ *and* $\Gamma \subseteq \Gamma'$ *with* Γ' *Context, then* $\Gamma' \vdash J$.
3. Substitution. *If* $\Gamma, x : A, \Gamma' \vdash J$ *and* $\Gamma \vdash N : A$, *then* $\Gamma', \Gamma[N/x] \vdash J[N/x]$.
4. Bound narrowing. *If* $\Gamma, x : A, \Gamma' \vdash J$ *and* $\Gamma \vdash A' \leq A$, *then* $\Gamma, x : A', \Gamma' \vdash J$.

The next proposition shows some implications between judgements. It is a common practice in the presentation of type theories to simply require the consequences of these as premises in the rules to begin with (often implicitly). Here our rules have fewer premises and we show the implications afterward, but our approach does make some proofs on derivations slightly harder, because sometimes we cannot apply an induction hypothesis directly. This can be circumvented by considering term structure instead, or subderivations of the premises.

Proposition 4.2 (Implied Judgements)

1. *If* $\Gamma \vdash J$ *then* Γ *Context.*

2. *If* $\Gamma \vdash J$ *and* $J \equiv M : A$, $A \leq B$ *or* $B \leq A$, *then* $\Gamma \vdash A$.

3. *If* $\Gamma \vdash J$ *and* $J \equiv M = N : A$, $N = M : A, \{M\}_A \leq B$ *or* $\{M\}_B \leq A$ *then* $\Gamma \vdash M : A$.

Generation principles are important for meta-theoretic analysis. They allow us to decompose a derived judgement into further derivable judgements concerning subterms from the first judgement. Typically a generation principle expresses the general way in which a judgement form may be constructed; for the context and type formation judgements, the generation principles are merely inversions of the rules. The following generation result for the subtyping judgement allows us to show generation for the typing judgement. It also reveals the "structural" nature of subtyping we mentioned before, except in the case of singletons: $\{M\}_A$ can be a subtype of a type B which is not itself a singleton. There is a case according to each syntactic form on either side of the subtyping symbol.

Proposition 4.3 (Subtyping Generation)

1. *If* $\Gamma \vdash P \leq B$ *then* B *is also an atomic type, say* P', *and* $P \leq_{Prim} P'$.

2. *If* $\Gamma \vdash \Pi x{:}\, A.\, B \leq C$ *then for some* A',B', *we have* $C \equiv \Pi x{:}\, A'.\, B'$, *such that (a)* $\Gamma \vdash A' \leq A$, *(b)* $\Gamma, x{:}\, A' \vdash B \leq B'$, *and (c)* $\Gamma, x{:}\, A \vdash B$.

3. *If* $\Gamma \vdash \{M\}_A \leq B$, *then* $\Gamma \vdash M : B$.

4. *If* $\Gamma \vdash A \leq P'$ *where* A *is not a singleton, then* A *is also an atomic type, say* P, *and* $P \leq_{Prim} P'$.

5. *If* $\Gamma \vdash C \leq \Pi x{:}\, A'.\, B'$ *where* C *is not a singleton, then for some* A,B, *we have* $C \equiv \Pi x{:}\, A.\, B$, *such that (a)* $\Gamma \vdash A' \leq A$, *(b)* $\Gamma, x{:}\, A' \vdash B \leq B'$, *and (c)* $\Gamma, x{:}\, A \vdash B$.

6. *If* $\Gamma \vdash C \leq \{N\}_B$ *then for some* A,M, *we have* $C \equiv \{M\}_A$.

Parts 3 and 6 of this proposition are rather weak, in particular nothing is said about the relation between types A and B. This will be rectified later.

The generation principle for the typing judgement $\Gamma \vdash M : A$ looks unusual, because we must account for the possibility that A is a singleton type.

Proposition 4.4 (Typing Generation)

1. *If* $\Gamma \vdash x : A$, *then* $\Gamma \vdash \{x\}_{\Gamma(x)} \leq A$

2. *If* $\Gamma \vdash \lambda x{:}\, A.\, M : C$, *then for some* A',B,B', *we have (a)* $\Gamma \vdash A' \leq A$, *(b)* $\Gamma,\, x{:}\, A' \vdash M : B'$, *(c)* $\Gamma,\, x{:}\, A \vdash M : B$, *and (d)* $\Gamma \vdash \{\lambda x{:}\, A.\, M\}_{\Pi x:A'.B'} \leq C$.

3. *If* $\Gamma \vdash MN : C$, *then for some* A, B, *we have that (a)* $\Gamma \vdash M : \Pi x{:}\, A.\, B$, *(b)* $\Gamma \vdash N : A$, *and (c)* $\Gamma \vdash \{MN\}_{B[N/x]} \leq C$.

In specific instances, the consequence of typing generation can be further broken down using the subtyping generation principle, and so on.

Admissible equality rules. We mention a few important admissible rules of $\lambda_{\leq\{\}}$. The symmetry and transitivity of equality

$$\frac{\Gamma \vdash M = N : A}{\Gamma \vdash N = M : A} \qquad (\text{EQ-SYM})$$

$$\frac{\Gamma \vdash L = M : A \qquad \Gamma \vdash M = N : A}{\Gamma \vdash L = N : A} \qquad (\text{EQ-TRANS})$$

are derived via (SUB-EQ-SYM) and (SUB-TRANS), using Proposition 4.2.

The usual rule for β-equality is (perhaps surprisingly) derivable. This is because $\lambda x{:}\, A.\, M$ can be given the tight dependent type $\Pi x{:}\, A.\, \{M\}_B$ using ($\{\}$-I), and so together with (λ) and (APP) we can derive:

$$\frac{\Gamma,\, x{:}\, A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x{:}\, A.\, M)N = M[N/x] : B[N/x]} \qquad (\text{EQ-}\beta)$$

This rule is used to show β subject reduction.

Removing Singletons. We mentioned that two parts of the subtyping generation principle (Proposition 4.3) are rather weak. If $\{M\}_A \leq B$, we would like to find a relationship between the types A and B. When B is not a singleton type, we expect (from the rules) that $A \leq B$. However, when $B \equiv \{N\}_C$ for some N and C, we may have $A \leq C$ or vice-versa, because of rules (SUB-EQ-SYM) and (SUB-EQ-ITER). A generation lemma covering these cases is untidy to state, and difficult to prove directly because of the rule (SUB-TRANS).

Here we define an operation $(-)^{\emptyset}$, which derives a non-singleton type from a type by repeatedly taking the type tag of a singleton type. A proposition relates a type to its singleton-deleted form; this sufficiently strengthens the generation result to give us a tool to show the admissibility of subject reduction and the minimal type property.

Definition 4.5 (Singleton Removal)

$$\begin{aligned} P^{\emptyset} &= P \\ (\Pi x{:}\, A.\, B)^{\emptyset} &= \Pi x{:}\, A.\, B \\ (\{M\}_A)^{\emptyset} &= A^{\emptyset} \end{aligned}$$

Proposition 4.6 (Singleton Removal Subtyping)

1. $\Gamma \vdash A \Longrightarrow \Gamma \vdash A \leq A^{\emptyset}$
2. $\Gamma \vdash A \leq B \Longrightarrow \Gamma \vdash A^{\emptyset} \leq B^{\emptyset}$
3. *If B is not a singleton, then* $\Gamma \vdash \{M\}_A \leq B \Longrightarrow \Gamma \vdash A \leq B$

Proof. Part 1 is proved by induction on the structure of types, part 2 by induction on the subtyping derivation, using 1 and Proposition 4.2; part 3 follows from 1 and 2 using the same proposition. □

Subject Reduction. We can now use the generation principles to show that β-subject reduction holds, for both typing and subtyping. The critical lemma is the case of a one-step outermost reduction. The syntactic proof of this is more involved than usual.

Theorem 4.7 (Subject Reduction)

1. *If $\Gamma \vdash M : A$ and $M \longrightarrow_\beta M'$, then $\Gamma \vdash M' : A$ also.*
2. *If $\Gamma \vdash A \leq B$ and $A \longrightarrow_\beta A'$, then $\Gamma \vdash A' \leq B$ also.*
3. *If $\Gamma \vdash A \leq B$ and $B \longrightarrow_\beta B'$, then $\Gamma \vdash A \leq B'$ also.*

Proof. Simultaneously for a single reduction step, we use induction on the structure of terms and types. For terms, this involves Proposition 4.4 and the equality rules, plus Lemma 4.8 below. For types, we use Proposition 4.3 and Proposition 4.2. □

Lemma 4.8 (Outermost β-reduction)

$\Gamma \vdash (\lambda x\colon A.\, M)\, N : C$ *implies* $\Gamma \vdash M[N/x] : C$.

Proof. By two applications of typing generation (Proposition 4.4), there exist types A_1, B_1, A_2, B_2 and B such that:

$$\begin{array}{ll}
\Gamma \vdash \lambda x\colon A.\, M : \Pi x\colon A_1.\, B_1, \qquad \Gamma \vdash N : A_1 & \\
\Gamma \vdash \{(\lambda x\colon A.\, M)\, N\}_{B_1[N/x]} \leq C & (*) \\
\Gamma \vdash A_2 \leq A, \qquad \Gamma, x\colon A \vdash M : B, \qquad \Gamma, x\colon A_2 \vdash M : B_2 & \\
\Gamma \vdash \{\lambda x\colon A.\, M\}_{\Pi x:A_2.B_2} \leq \Pi x\colon A_1.\, B_1 &
\end{array}$$

By Propositions 4.6 and 4.3 and the last of these, we have:

$$\Gamma \vdash \Pi x\colon A_2.\, B_2 \leq \Pi x\colon A_1.\, B_1, \qquad \Gamma \vdash A_1 \leq A_2, \qquad \Gamma, x\colon A_1 \vdash B_2 \leq B_1.$$

Now using (BND-NARROW) and (SUB) we have $\Gamma, x\colon A_1 \vdash M : B_1$. So we can apply the admissible rule (EQ-β),

$$\frac{\Gamma, x\colon A_1 \vdash M : B_1 \qquad \Gamma \vdash N : A_1}{\Gamma \vdash (\lambda x\colon A_1.\, M)\, N = M[N/x] : B_1[N/x]}$$

and by (EQ-λ) and (EQ-APP):

$$\frac{\Gamma \vdash A_1 \leq A \qquad \Gamma, x{:}\, A_1 \vdash M = M : B_1 \qquad \Gamma, x{:}\, A \vdash M : B \qquad \Gamma \vdash N : A_1}{\Gamma \vdash (\lambda x{:}\, A.\, M)\, N = (\lambda x{:}\, A_1.\, M)\, N \;\; : B_1[N/x]}$$

By transitivity:

$$\Gamma \vdash (\lambda x{:}\, A.\, M)\, N \; = \; M[N/x] \;\; : B_1[N/x] \tag{$\dagger$}$$

Finally, using Proposition 4.2, (SUB-TRANS), and (SUB-EQ-SYM), we can derive:

$$\frac{\dfrac{\Gamma \vdash P = Q : D}{\Gamma \vdash \{Q\}_D \leq \{P\}_D} \qquad \Gamma \vdash \{P\}_D \leq C}{\Gamma \vdash Q : C}$$

Let ($\dagger$) and (*) be the premises, so $P \equiv (\lambda x{:}\, A.\, M)\, N$, $Q \equiv M[N/x]$, and $D \equiv B_1[N/x]$. Then $\Gamma \vdash M[N/x] : C$ as required. $\square$

Minimal Types. If a term possesses several types, it is useful both theoretically and pragmatically if one type can always be found that is more general than the others; in subtyping systems, it is *minimal*. With untagged singletons, minimal types are a triviality: the minimal type for a term M is $\{M\}$! When type tags are added, the issue is not so obvious. Here we show a strengthening of typing generation to give minimal types.

The minimal type $\min_\Gamma(M)$, of a term M in a context Γ, has the form $\{M\}_A$ for some A. We give a partial inductive definition of $\min_\Gamma(M)$ which is shown in the following lemma to be well defined on all Γ, M such that $\Gamma \vdash M : A$ for some A.

Definition 4.9 (Minimal Types)

$$\begin{array}{rll}
\min_\Gamma(x) & = \{x\}_{\Gamma(x)} & \\
\min_\Gamma(\lambda x{:}\, A.\, M) & = \{\lambda x{:}\, A.\, M\}_{\Pi x : A. \min_{\Gamma, x:A}(M)} & \\
\min_\Gamma(MN) & = \{MN\}_{B[N/x]} \quad \textit{where} & (\min_\Gamma(M))^{\{\!\!/\!\}} \equiv \Pi x{:}\, A.\, B \\
& & \textit{and}\ \Gamma \vdash N : A
\end{array}$$

Lemma 4.10 (Existence of Minimal Types)

1. $\Gamma \vdash M : A \qquad \Longrightarrow \qquad \Gamma \vdash M : \min_\Gamma(M)$

2. $\Gamma \vdash M : A \qquad \Longrightarrow \qquad \Gamma \vdash \min_\Gamma(M) \leq A$

Proof. Simultaneously, by induction on the derivation of $\Gamma \vdash M : A$. $\square$

Minimal types are not unique, and the minimal types given by the simple definition of $\min_\Gamma(M)$ are not necessarily the simplest syntactically. For example, if $\Gamma \equiv z{:}\, \alpha, f{:}\, \beta \to \{z\}_\alpha, x{:}\, \beta$ then $\min_\Gamma(fx) \equiv \{fx\}_{\{z\}_\alpha}$. But $\Gamma \vdash fx : \{z\}_\alpha$ too, and we can show that $\Gamma \vdash \{z\}_\alpha \leq \{fx\}_{\{z\}_\alpha}$.

5 A PER Interpretation of $\lambda_{\leq\{\}}$

Subtyping calculi have two basic kinds of model. We may choose a *typed* value space where subsumption is modelled using coercion maps between types. In some sense this is the most general setting, but it requires some way of relating coercion maps to the syntax: either we forgo (SUB) and introduce coercions explicitly into the syntax [CL91], or we reconstruct coercions by some translation process [BTCGS91]. Either route requires a *coherence* property of the interpretation, because of the possibility of different ways of deriving or expressing the analogue of a coercion-free statement, by permuting the positions of coercions. This property can be quite tricky to establish in a general form, and has yet to be demonstrated in a subtyping calculus more complex than $F_{\leq}$ (see [CG92]). Here we follow the alternative *untyped* approach, based on a global value space from which types are carved out. Coercion maps are unnecessary since subtyping amounts to inclusion between types, and the interpretation of a term does not depend on its type. The need for coherency properties can be avoided by defining the interpretation by induction on the structure of raw (coercion-free) terms rather than typing derivations.

The PER model. Recall that a partial equivalence relation (PER) on a set D is a symmetric and transitive relation $R \subseteq D \times D$. The domain of R, $dom(R)$, is the set $\{d \mid d\ R\ d\}$, but we often write $d \in R$ instead of $d \in dom(R)$. The equivalence class $\{d' \mid d'\ R\ d\}$ of d in R is written $[d]_R$. Subtyping will be interpreted as inclusion of PERs, which is simply subset inclusion on $D \times D$.

The construction is mostly standard (see e.g., [CL91, BL90]), but incorporates type-term dependency. We make use of a model of the untyped λ-calculus to interpret terms and to build PERs over.

Definition 5.1 (Lambda Model [HL80])
A lambda model *is a triple, $\mathcal{D} = \langle D, \cdot, [\![\ ]\!] \rangle$, where D is a set, $\cdot$ is a binary operation on D and for untyped lambda-terms M, the interpretation of M in an environment $\rho\colon Var \rightharpoonup D$ is $[\![M]\!]_\rho \in D$, such that:*

$$[\![x]\!]_\rho = \rho(x) \tag{VAR}$$

$$[\![MN]\!]_\rho = [\![M]\!]_\rho \cdot [\![N]\!]_\rho \tag{APP}$$

$$[\![\lambda x.\, M]\!]_\rho = [\![\lambda y.\, M[y/x]]\!]_\rho \tag{α}$$

$$(\forall d \in D.\ [\![M]\!]_{\rho[x \mapsto d]} = [\![N]\!]_{\rho[x \mapsto d]}) \implies [\![\lambda x.\, M]\!]_\rho = [\![\lambda x.\, N]\!]_\rho \tag{ξ}$$

$$(\forall x \in FV(M).\ \rho(x) = \rho'(x)) \implies [\![M]\!]_\rho = [\![M]\!]_{\rho'} \tag{FV}$$

$$\forall d \in D.\ [\![\lambda x.\, M]\!]_\rho \cdot d = [\![M]\!]_{\rho[x \mapsto d]} \tag{β}$$

From the above axioms (except β), we also have:

$$[\![M[N/x]]\!]_\rho = [\![M]\!]_{\rho[x \mapsto [\![N]\!]_\rho]} \tag{SUBSTITUTE}$$

An environment ρ' extends another ρ, written $\rho \subseteq \rho'$, if for all variables x, if $\rho(x)$ is defined then $\rho'(x)$ is defined and $\rho(x) = \rho'(x)$. Fix a lambda-model $\mathcal{D}$ with domain D. Terms are interpreted as elements of D as usual (we leave the *erase* operation, which deletes type information, implicit) and types are interpreted as PERs on D.

Pairing and projection operations in the model are defined by:

$$\begin{aligned}
\langle a, b\rangle &= [\![\lambda f.\, fxy]\!]_{[x \mapsto a,\, y \mapsto b]} \\
\pi_1 p &= p \cdot [\![\lambda x.\, \lambda y.\, x]\!]_{[\,]} \\
\pi_2 p &= p \cdot [\![\lambda x.\, \lambda y.\, y]\!]_{[\,]}
\end{aligned}$$

We first show some constructions for building PERs, and then the interpretation proper.

Definition 5.2 (PER Constructions)
We define PERs to interpret the types of $\lambda_{\leq\{\}}$, as follows:

- *For each primitive type P, we assume a PER R_P such that $P \leq_{Prim} P'$ implies $R_P \subseteq R_{P'}$.*
- *Let R be a PER and $S(a)$ be a PER for all $a \in dom(R)$, such that $S(a) = S(b)$ whenever $a \;\; R \;\; b$.*

  Define the PER $\Pi(R, S)$ by:

  $$f \;\; \Pi(R,S) \;\; g \qquad \text{iff} \qquad \forall a, b.\;\; a \;\; R \;\; b \implies f \cdot a \;\; S(a) \;\; g \cdot b$$

  Define the PER $\Sigma(R, S)$ by:

  $$\langle a_1, b_1\rangle \;\; \Sigma(R,S) \;\; \langle a_2, b_2\rangle \qquad \text{iff} \qquad a_1 \, R \, a_2 \quad \text{and} \quad b_1 \;\; S(a_1) \;\; b_2$$

- *Let R be a PER. Define the PER $[p]_R$ by:*

  $$m \;\; [p]_R \;\; n \qquad \text{iff} \qquad m \, R \, n \quad \text{and} \quad m \, R \, p$$

The interpretation $[\![\Gamma]\!]$ of a context Γ is a PER. The interpretation of a type in some context is a family of PERs $[\![\Gamma \vdash A]\!]_g$ indexed by elements $g \in [\![\Gamma]\!]$ that is invariant under the choice of representative of equivalence class in $[\![\Gamma]\!]$.

Definition 5.3 (Interpretation of Contexts and Types)
For each context Γ, we define a PER $[\![\Gamma]\!]$ by:

$$[\![\langle\rangle]\!] = D \times D$$

$$[\![\Gamma, x{:}\, A]\!] = \Sigma([\![\Gamma]\!], [\![\Gamma \vdash A]\!])$$

For each context Γ *and type* A*, we define a PER* $[\![\Gamma \vdash A]\!]_g$*, for each* $g \in dom([\![\Gamma]\!])$*:*

$$[\![\Gamma \vdash P]\!]_g = R_P$$

$$[\![\Gamma \vdash \Pi x\colon A.\, B]\!]_g = \Pi([\![\Gamma \vdash A]\!]_g, \Lambda a.\ [\![\Gamma, x\colon A \vdash B]\!]_{\langle g,\, a\rangle})$$

$$[\![\Gamma \vdash \{M\}_A]\!]_g = [\ [\![M]\!]_{g^\Gamma}\ ]_{[\![\Gamma \vdash A]\!]_g}$$

Notice that $\vdash$ is just used as a place-holder here, it does not signify a judgement derivation. The symbol Λ stands for lambda-abstraction at the meta-level, and $g^\Gamma\colon Var \rightharpoonup D$ is the environment defined by projections on g:

$$g^{\langle\rangle}(y) \text{ undefined, for all } y.$$

$$g^{\Gamma,\, x:A}(y) = \begin{cases} \pi_2(g), & \text{if } y \equiv x, \\ (\pi_1(g))^\Gamma(y) & \text{if } y \not\equiv x. \end{cases}$$

The following theorem establishes the main soundness property of the model construction. It also shows well-definedness of the interpretation of contexts $[\![\Gamma]\!]$ and of types $[\![\Gamma \vdash A]\!]$, whenever Γ is a context and $\Gamma \vdash A$. The parts of the theorem need to be proven simultaneously because of type dependency.

Theorem 5.4 (Well-Definedness and Soundness)

1. *If* Γ *Context then* $[\![\Gamma]\!]$ *is a PER.*
2. *If* $\Gamma \vdash A$ *then* $[\![\Gamma \vdash A]\!]_{g_1} = [\![\Gamma \vdash A]\!]_{g_2}$ *whenever* $g_1\ \ [\![\Gamma]\!]\ \ g_2$.
3. *If* $\Gamma \vdash M : A$ *then* $[\![M]\!]_{g_1{}^\Gamma}\ \ ([\![\Gamma \vdash A]\!]_{g_1})\ \ [\![M]\!]_{g_2{}^\Gamma}$ *whenever* $g_1\ \ [\![\Gamma]\!]\ \ g_2$.
4. *If* $\Gamma \vdash A \leq B$ *then* $[\![\Gamma \vdash A]\!]_g \subseteq [\![\Gamma \vdash B]\!]_g$ *for all* $g \in [\![\Gamma]\!]$.

Proof. Simultaneously by induction on derivations, making use of axioms of the lambda-model and the definitions of the PER constructions. □

Notice that a special case of part 3 is the soundness property that $\Gamma \vdash M : A$ implies $[\![M]\!]_{g^\Gamma} \in [\![\Gamma \vdash A]\!]_g$ for any $g \in [\![\Gamma]\!]$. The existence of PER models (generated from λ-models and some PERs to interpret atomic types), together with this soundness result, guarantee the consistency of the calculus.

6 Related and Further Work

We have presented the type system $\lambda_{\leq\{\}}$, which adds singleton types and Π-types to the simply-typed lambda-calculus with subtypes, $\lambda_{\leq}$. This gives a system with types that depend on terms. Singleton types $\{M\}_A$ can be interpreted in a PER model as the equivalence class of $[\![M]\!]$ in the PER $[\![A]\!]$.

The judgements of $\lambda_{\leq\{\}}$ are defined simultaneously and typing embeds in subtyping in a simple way. Noticing this, we can give alternative shorter presentations of the system which "wrap-up" judgements in terms of each other; this can be useful because it simplifies tedious simultaneous inductions. One way is to encode typing in terms of subtyping, defining $\Gamma \vdash M : A$ iff $\Gamma \vdash \{M\}_A \leq B$ holds for some B. Another is to encode subtyping in terms of typing, defining $\Gamma \vdash A \leq B$ iff $\Gamma, x{:}A \vdash x : B$; we get the usual subtyping rules if a rule of η-reduction is added.

Decidability of type-checking has not yet been investigated, but we conjecture that it holds. A typical approach would be to first study the reduction behaviour of terms and types, for both β-reduction and Γ-reduction. We would hope to find a normalization result and a set of syntax-directed rules for the system, with points where normalization occurs. The final step would be to give a termination argument for the deterministic rules, to show that they define a complete algorithm.

Research into type systems for object-oriented programming (e.g., [Car88b, BL90, CG92]) gave inspiration for this work. Most systems in the literature do not treat type dependency at the same time as subtyping (Cardelli's is an exception). Hayashi's extensions of System F [Hay94] have singleton types with the same form as those described here. His calculi have union and intersection types, so dependent products become a derived notion. Hayashi's treatment of singletons differs; his is based on encoding a propositional equality (which reflects his intention to create a constructive logic), rather than a definitional one. He has to add a rule of subject reduction as primitive, since there is a counterexample to its admissibility involving union types.

We hope that some of the development here will carry over to systems with application at the level of types, but the situation with full dependent types is more difficult than the simple case where each term in a type is warmly insulated with singleton braces. The harder case is the topic of some joint work in progress between the author and Adriana Compagnoni.

Acknowledgements. I am grateful to Benjamin Pierce, Don Sannella, and Andrzej Tarlecki for their wisdom, help, and encouragement during the progress of this work. Useful comments and corrections were provided by the CSL referees. I was supported by a UK EPSRC postgraduate studentship.

References

[Asp95] David R. Aspinall. Algebraic specification in a type-theoretic setting. Forthcoming PhD thesis, Department of Computer Science, University of Edinburgh, 1995.

[BL90] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87:196–240, 1990.

[BTCGS91] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[Car88a] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car88b] Luca Cardelli. Structural subtyping and the notion of power type. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988.

[CG92] 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.

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[Hay94] Susumu Hayashi. Singleton, union and intersection types for program extraction. *Information and Computation*, 109, 1994.

[HL80] R. Hindley and G. Longo. Lambda calculus models and extensionality. *Z. Math. Logik Grundlag. Math.*, 26:289–310, 1980.

[HP91] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[KST94] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, LFCS, Department of Computer Science, University of Edinburgh, 1994.

[SP94] Paula Severi and Erik Poll. Pure Type Systems with Definitions. In *Logical Foundations of Computer Science, LFCS'94*, Lecture Notes in Computer Science 813, pages 316–328. Springer-Verlag, 1994.

[SST92] Donald T. Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

[SW83] Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation. In *Proceedings of International Conference on Foundations of Computation Theory, Borgholm, Sweden*, Lecture Notes in Computer Science 158. Springer-Verlag, 1983.