Type Inference and Implicit Scaling

Satish Thatte
Department of Mathematics and Computer Science
Clarkson University, Potsdam, NY 13676, USA

Abstract

We describe a novel application of subtyping in which a small orthogonal set of structural subtyping rules are used to capture the notion of scaling—an unusual variety of polymorphism found in APL-like languages which is attracting renewed interest due to its applications in data parallel programming. The subtyping approach allows us to provide a simple coercion-based semantics for a generalized interpretation of scaling that goes well beyond what is available in APL dialects.

1 Introduction

Data parallelism [HS86,Bre88,Vis89] has gained increasing favor recently due to its conceptual simplicity and the high speedup available whenever the technique can be used effectively. Conceptually, data parallelism covers a broad range from traditional vector processing to techniques for programming Transputer networks [Vis89] and the Connection Machine [HS86]. In this paper, we are concerned with one of the main techniques used for data parallel programming: the technique of *scaling*, which goes back all the way to APL [Ive62] where it was introduced for its expressive power in array manipulation rather than as a way of expressing parallelism. Our concern will be with the implications of scaling for static typing—specifically, we explore a novel subtyping approach to the static type analysis of a very general interpretation of scaling.

Recall that in APL, many scalar operations also accept array arguments and "scale" their meaning accordingly. In later dialects like APL2, the arguments may also be arbitrarily nested arrays. For instance, the scaling and shifting of a vector is usually written as $a+b\vec{v}$ where $\vec{v}$ is a vector and a and b are real constants. Representing $\vec{v}$ by a 1-D array V, one can simply write this expression as `a+b*V` in APL. In Standard ML [Mil84], using a list V, the same expression might be written as `map (op +) (distl (a, map (op *) (distl (b, V))))` where the `distl` primitive is borrowed from FP [Bac78]. Besides the obvious implicit parallelism, the gain in expressive power as a result of scaling is striking.

The price paid for implicit scaling is added complexity in the semantics of the language. Existing explanations of scaling in APL [Ben85,JM78] treat only the operations involved (such as "+" and "*" above) as being polymorphic. The range of possible behaviors of such operations, especially when nested structures are allowed as arguments, is hard to capture in a single principle type expression, or even in a finite number of expressions. This is the main difficulty in doing static type analysis of scaled expressions. Our innovation in this paper is to show that an alternative approach based on coercive structural subtyping accounts very effectively for scaling. In effect, our type system coerces the APL-like version of the expression given above to the Standard ML version. We expect that a realistic compiler using our system can derive enough information from the typing process to generate more efficient (sequential or data parallel) code than the naive synthesized version implies.

The generality achieved by our solution goes well beyond what is available in APL dialects. Scaling is no longer limited to syntactic operators—all functions including user-defined ones can be scaled up in the same way. The extension of the subtype structure relative to type constructors captures all the natural implications of scaling for components of structures, higher-order functions, and so forth (see examples in Section 3). The notion of scaling itself is more general. As an example, suppose "++" denotes vector concatenation, and the vector consisting of $x_1,x_2,\ldots,x_n$ is denoted by $[x_1,x_2,\ldots,x_n]$. The expressions [[1,2],[3,4]] ++ [[5,6],[7,8]] and [[1,2],[3,4]] + [[5,6],[7,8]] both work correctly: the former yields [[1,2,5,6], [3,4,7,8]] and the latter [[6,8], [10,12]]. Note that the grain of scaling is different in the two cases. We do not know of any APL dialect which *automatically* adjusts the grain of scaling to the needs of the application in this way. Our technique is also quite robust under many kinds of enrichments of the underlying language—for instance with mutable variables. Compatibility with parametric polymorphism poses some interesting problems, which are discussed in Section 8.

The basis of our solution is a small set of orthogonal subtyping rules (with corresponding coercions) which capture most cases of scaling. As in the case of subtyping with labeled record types [Car88] subtyping is based on the structure of type expressions. Although easy to understand and motivate, the structural relationships turn out to be unusually complex. Even the antisymmetry of the subtype relation needs a nontrivial proof. The proof of the coherence of subsumptions (subtyping judgements), *i.e.*, the property that each subsumption implies a semantically unique abstract coercion, requires a normalization result for derivations of subsumptions. The subtype structure is consistently complete, but this is not obvious, and the algorithms for finding LUBs and GLBs (required in the typechecking algorithm) are quite complex. In spite of this complexity, we believe that the subtype structure is intuitively natural and will be "user-friendly" in practice.

The subtyping rules define the rest of the problem, which is to verify that they can be applied within a standard general framework of the kind given in [Rey85] to give unambiguous meanings to scaled expressions. Standard typing rules allow derivation of types and coerced (unscaled) versions for all meaningful scaled expressions and each coerced version can be given a meaning using the standard semantics of the λ-calculus. To show that each scaled expression has a *unique* meaning, we need two further properties: the existence of a minimal typing judgement for each well-typed expression, and semantic coherence—the property that the meaning of an expression depends only on the typing judgement applied to it, not on the derivation used to reach that judgement. Since each use of a subsumption in a typing derivation implies the insertion of a coercion, the meaning of an expression seems to depend on the particular derivation. Coherence asserts that this apparent ambiguity is semantically inconsequential: all the different coerced versions for the same judgement have the same meaning. The notion of coherence was first discussed explicitly in [BC$^+$89]. Reynolds' discussion of coercions and overloaded operators [Rey85] is based on the same intuition. As Reynolds (implicitly) points out, coherence of typing is closely related to coherence of subsumptions. The additional complication in our case comes from the fact that each function-valued expression is "overloaded" with an infinite number of potential meanings. However, it can be shown that at most one of these overloaded meanings is usable in any particular application. This fact, together with coherence of subsumptions, turns out to be sufficient for coherence of typing.

In the rest of the paper, following a brief discussion of related work and some preliminaries in Sections 2 and 3, we begin by deriving the subtype structure in Section 4. Section 5 gives an outline of the coercion-based semantics. The proofs of the major properties of the subtype

$e ::= x$	(identifiers)	$\mid \lambda x_\tau.\, e$	(typed abstractions)	$\mid e_1\ e_2$		(applications)	
$\mid e_1, e_2$	(pairs)	$\mid e \downarrow i$	(projections, i=1,2)	$\mid \underline{\text{nil}}_\tau$		(empty list)	
$\mid e_1; e_2$	(cons)	$\mid \underline{\text{hd}}\ e$	(list head)	$\mid \underline{\text{tl}}\ e$		(list tail)	
$\tau ::= \iota$	(scalar types)	$\mid \tau_1 \times \tau_2$	(product types)				
$\mid [\tau]$	(list types)	$\mid \tau_1 \rightarrow \tau_2$	(function types)				

Figure 1: Syntax of Object and Type Expressions

structure are outlined in Section 6. Section 7 gives the typing algorithms, and Section 8 concludes with a discussion of the problems involved in adding parametric polymorphism. Many technical details and all actual proofs are omitted in this version for lack of space.

2 Related Work

Type inference using subtypes structures has proved to be a fruitful idea in a variety of applications. It was originally introduced by Reynolds [Rey80] to systematize the semantics of automatic coercions between types. Such subtyping might be called *coercive*, to contrast it with the *inclusive* variant used in theories of inheritance [Car88], quantified types [Mit88] and partial types [Tha88], where subtypes are taken to be subsets. Most applications of the coercive variant have been concerned with relationships between *atomic* types, such as "integer $\leq$ real". An underlying theme in this paper is that coercive *structural* subtyping—subtyping based on the structure of type expressions—can be very useful as a tool to provide coercion based semantics for many interesting language features that pose problems for other semantic approaches. A similar approach is used in [BC$^+$89] to give an alternative semantics for inheritance. We have elsewhere [Tha90] explored an application to dynamic typing in static languages.

3 Type and Object Languages

The object language is a simply typed dialect of the λ-calculus. For definiteness, the language includes a linear list or sequence structure for the application of scaling. However, this fact is nowhere used in an essential way, and substituting sequences with any other data structure suitable for set representation (such as trees or arrays) would require no change in the treatment except for the substitution of appropriate new conversion functions. The grammars for type and object expressions are given in Figure 1, where the metavariable e ranges over expressions, x over identifiers, ι over scalar types and τ over all type expressions. Scalar types in this context need not include only atomic types. Any type which is not a product or function type and is not a structure type involved with scaling can be thought of as a scalar type. The set of all type expressions will be denoted by *Typexprs*.

Besides the constructors $\times$ and $\rightarrow$ for product and function types, we have an outfix type constructor [ ]; $[\tau]$ is list-of-τ. We need to provide the list primitives as syntactic operators in order to allow them to be generic. Note that the type intended for each use of $\underline{\text{nil}}$ must be given (this can be avoided by introducing the "universal" type described by Reynolds [Rey85]). In a simply typed dialect of the λ-calculus such as ours, recursion must normally be provided by an explicit construct which computes least fixed points of functions. The reason for omitting the construct in the grammar above is that fixpoint constructs are incompatible with minimal typing in our context—the counterexample is omitted here for lack of space. This does *not* mean that the language cannot include fixpoint constructs. It does mean that the typing constraints for such constructs cannot be described using nondeterministic typing rules as in the case of the other constructs. It is easy to infer the *natural* type of instances of the fix construct, and the fix case in the minimal typing algorithm **Type** in Section 7 does exactly that.

4 The Subtype Structure

The essence of our approach is to capture the semantics of scaling in a small orthogonal set of structural subtyping rules. The subtype structure must find a balance between two conflicting principles—orthogonality and coherence. Orthogonality—the treatment of all (data and function) types as first-class citizens in the subtyping scheme—is what gives the solution its simplicity, generality and expressive power. Unrestricted orthogonality leads to loss of coherence, but the coherent solution derived below retains sufficient orthogonality for most practical purposes.

It is helpful to start with some examples to outline the desired range of applicability of the subtype structure. The primitive coercions we shall need are provided in FP [Bac78] as primitives—"•" (function composition), α (a *curried* version of map), distl, distr and trans. The function trans transforms any pair of lists of equal size into a list of pairs of corresponding elements in the obvious way; distl "distributes" its first argument by pairing it with elements of its second (list/sequence) argument, and distr is exactly the same except it takes its arguments in the reverse order. We treat these coercions as though they possess polymorphic types because they are used only in places where their type is both correct and manifest. The use of FP primitives as basic coercions is especially interesting because FP has been influenced by many APL ideas and idioms but lacks a notion of scaling. The reason (presumably) is that the semantics of implicit scaling in APL is rather complex and *ad hoc*. We restore scaling (for homogeneous structures) in a semantically simple way by *implicitly* using the same coercions FP programmers must use *explicitly*.

We use $[e_1, e_2, \ldots, e_n]$ as an abbreviation for $e_1; (e_2; (\ldots; (e_n; \underline{\text{nil}}_\tau) \ldots))$ (where τ is the component type) and the form $e \twoheadrightarrow e'$ to mean that the expression e is (expected to be) coerced to e' by a minimal typing derivation. Thus,

$$\begin{aligned}
\text{square}\,[1,2,3] &\twoheadrightarrow (\alpha\,\text{square})\,[1,2,3] = [1,4,9]\\
1 + [1,2,3] &\twoheadrightarrow (\alpha +)\,(\text{distl}\,(1,[1,2,3])) = [2,3,4]\\
[1,2,3] + 1 &\twoheadrightarrow (\alpha +)\,(\text{distr}\,([1,2,3],1)) = [2,3,4]\\
[1,2,3] + [2,3,4] &\twoheadrightarrow (\alpha +)\,(\text{trans}\,([1,2,3],[2,3,4])) = [3,5,7]
\end{aligned}$$

Scaling is not limited to one "level" in a structure. Thus,

$$1 + [[1,2],[2,3]] \twoheadrightarrow (\alpha\,(\alpha +))\,((\alpha\,\text{distl})\,(\text{distl}\,(1,[[1,2],[2,3]]))) = [[2,3],[3,4]]$$

For an example with nonscalar operands, let $f = \lambda x_{\text{int}\times[\text{int}]}.\, x{\downarrow}1; x{\downarrow}2$,

$$f\,(0,[[1,2],[2,3]]) \twoheadrightarrow (\alpha f)\,(\text{distl}\,(0,[[1,2],[2,3]])) = [[0,1,2],[0,2,3]]$$

We wish to capture the implicit coercions implied by these examples in a few orthogonal structural subtyping rules. Subtyping judgements will be presented in the "natural deduction" style. Each subtyping judgement has the form $\vdash \tau 1 \le \tau 2 \Rightarrow f$ where f is the corresponding coercion. The simple scaling of functions as in square [1, 2, 3] can be captured in its full generality by the rule

$$\text{SCL:}\quad \vdash \quad \tau 1 \to \tau 2 \;\le\; [\tau 1] \to [\tau 2] \quad \Rightarrow \quad \alpha$$

which uses the (polymorphic) operator α to convert any function of type $\tau 1 \to \tau 2$ to a function of type $[\tau 1] \to [\tau 2]$, where $\tau 1$ and $\tau 2$ are arbitrary types. For instance, consider the expression square [[1, 2], [2, 3]]. Here the type of square is coerced to [[int]] $\to$ [[int]] by two iterations of SCL, and square itself is coerced to α (α square). An interesting consequence of SCL is that one *never* needs to use the α (map) operator explicitly, even in order to scale up an argument of a higher-order function (see inner product example at the end of the section).

Evaluation of expressions like [1, 2, 3] + [2, 3, 4] can be seen as a two step process in which a *zipping* step collates the two operands to yield [(1,2), (2,3), (3,4)] and a *scaling* step coerces "+" to "α +". The first step can be captured by the rule

ZIP: $\vdash \quad [\tau 1] \times [\tau 2] \ \leq\ [\tau 1 \times \tau 2] \quad \Rightarrow \quad$ trans

with the semantic proviso (enforced by trans) that the two lists must have the same length. This generalizes pleasantly to examples like

$$\begin{aligned} & [[1, 2], [3, 4]] + [[2, 3], [4, 5]] \\ \twoheadrightarrow\ & (\alpha\,(\alpha +))\ ((\alpha\,\text{trans}) \bullet \text{trans}\ ([[1, 2], [3, 4]], [[2, 3], [4, 5]])) \ = \ [[3, 5], [7, 9]] \end{aligned}$$

The argument type [[int]] $\times$ [[int]] is transformed to [[int $\times$ int]] by two iterations of ZIP, and "+" is then applicable by two iterations of SCL. The second iteration of ZIP uses a naturally induced subtyping relationship between list types (incorporated into rule LIST in Figure 2). ZIP implies that all explicit uses of our version of trans can also be eliminated.

This leaves examples like 1 + [1, 2, 3]. The argument type here is int $\times$ [int] and it needs to be subsumed to [int $\times$ int]. The coercion involves *replication* of the first argument to match the second. Replication cannot be separated from zipping since the degree of replication is determined by the context—1 is replicated three times in this example because the *other* argument of "+" is a list of length three. We might therefore propose the symmetric rules

$\vdash \ \tau 1 \times [\tau 2] \ \leq\ [\tau 1 \times \tau 2] \ \Rightarrow\ $ distl $\quad$ and $\quad \vdash\ [\tau 1] \times \tau 2 \ \leq\ [\tau 1 \times \tau 2] \ \Rightarrow\ $ distr

Unfortunately, these rules are incompatible with coherence. The problem can be seen with a simple example — two semantically distinct derivations for [int] $\times$ [int] $\leq$ [[int $\times$ int]]:

[int] $\times$ [int] $\leq$ [int $\times$ [int]] $\leq$ [[int $\times$ int]] $\qquad\qquad$ [int] $\times$ [int] $\leq$ [[int] $\times$ int] $\leq$ [[int $\times$ int]]

The coercion for [int] $\times$ [int] $\leq$ [[int $\times$ int]] is distl $\bullet$ distr in the first derivation, and distr $\bullet$ distl in the second: ([1,2],[3,4]) would be converted to [[(1,3),(1,4)], [(2,3),(2,4)]] by the first derivation and to [[(1,3),(2,3)], [(1,4),(2,4)]] by the second. We therefore impose the restriction that replicated values must be *scalars*.

The basic cases of the subtype relation are defined by rules SCL, ZIP, REPL and REPR in Figure 2. The other rules in Figure 2 are standard for all subtype relations (see, *e.g.*, [Rey85]). Of these, LIST, PROD, and FUN allow the basic rules to be applied to *subexpressions* of a type expression in a natural way. In the coercion for PROD, we have used FP's selection functions **1** and **2** as projections from pairs, and FP's *construction* form in its dyadic version—the construction "$\{f_1, f_2\}$" denotes a function such that $\{f_1, f_2\}\ x \ =\ (f_1\ x, f_2\ x)$.

Clearly, the coercions in Figure 2 are naive. In a serious sequential implementation, one would expect to optimize the implementation of standard combinations to avoid actual zipping and replication whenever possible, to produce code that is comparable in efficiency to (say) equivalent hand-coded C programs. In programming for the Connection Machine on the other hand, actual replication appears to be the standard practice [HS86]. The detection and

SCL: $\vdash \tau_1 \to \tau_2 \le [\tau_1] \to [\tau_2] \Rightarrow \alpha$ ZIP: $\vdash [\tau_1] \times [\tau_2] \le [\tau_1 \times \tau_2] \Rightarrow \mathsf{trans}$

REPL: $\vdash \iota \times [\tau] \le [\iota \times \tau] \Rightarrow \mathsf{distl}$ REPR: $\vdash [\tau] \times \iota \le [\tau \times \iota] \Rightarrow \mathsf{distr}$

RFLX: $\vdash \tau \le \tau \Rightarrow \mathsf{id}$

TRNS: $$\frac{\vdash \tau1 \le \tau2 \Rightarrow f \qquad \vdash \tau2 \le \tau3 \Rightarrow g}{\vdash \tau1 \le \tau3 \Rightarrow g \bullet f}$$

LIST: $$\frac{\vdash \tau1 \le \tau2 \Rightarrow f}{\vdash [\tau1] \le [\tau2] \Rightarrow \alpha f}$$

PROD: $$\frac{\vdash \tau1 \le \tau2 \Rightarrow f \qquad \vdash \tau3 \le \tau4 \Rightarrow g}{\vdash \tau1 \times \tau3 \le \tau2 \times \tau4 \Rightarrow \{f \bullet \mathbf{1},\ g \bullet \mathbf{2}\}}$$

FUN: $$\frac{\vdash \tau1 \le \tau2 \Rightarrow f \qquad \vdash \tau3 \le \tau4 \Rightarrow g}{\vdash \tau2 \to \tau3 \le \tau1 \to \tau4 \Rightarrow \lambda h.\ g \bullet h \bullet f}$$

Figure 2: Subtyping Rules and Coercions

transformation of optimizable combinations of coercions can be made a part of the typechecking algorithm. The details are clearly nontrivial, and will have to await another paper.

To illustrate the use of a number of rules working together, consider a slightly more complex example involving higher-order functions. In FP, the inner product function is defined by the expression (/+) • (α∗) • trans, where "/" is APL's *reduce* operator, which has type (real × real → real)→[real]→real in this context. Given that explicit uses of α and trans are unnecessary, we should be able to express inner product as (/+) • ∗. The expression should have the type τ = [real] × [real] → real. "/+" clearly has type [real]→real. The type of "∗" is coerced from real × real → real to [real × real] → [real] using SCL to fit the composition, giving the (minimal) type σ = [real × real] → real for the overall expression. It is easy to see that the required type τ is a supertype of σ—[real] × [real] ≤ [real × real] by ZIP and hence $\tau \geq \sigma$ by FUN. The standard behavior is therefore *inherited* by our version, which is more general than the usual inner product. In addition to a pair of real sequences, it could also be applied to a real constant and real sequence, or to a sequence of real pairs.

The subtype structure defined here appears to have few unexpected consequences of the kind that made coercions in PL/I notorious. A possible exception is that some nonhomogeneous list expressions, instead of producing type errors, are automatically homogeneized:

[(3, [1,2]), ([4,5], 6)] ↠ [distl (3, [1,2]), distr ([4,5], 6)] = [[(3,1),(3,2)], [(4,6),(5,6)]]

It should be noted that in all of the examples in this section, whenever automatic coercion is required, the resulting converted expression is not unique. Given an apparently mismatched

$$A \vdash x \Rightarrow x : A(x)$$

$$\frac{A \vdash e_1 \Rightarrow e_1' : \tau \qquad A \vdash e_2 \Rightarrow e_2' : [\tau]}{A \vdash e_1; e_2 \Rightarrow e_1'; e_2' : [\tau]}$$

$$\frac{A \vdash e \Rightarrow e' : [\tau]}{A \vdash \underline{\mathrm{hd}}\ e \Rightarrow \underline{\mathrm{hd}}\ e' : \tau}$$

$$\frac{A + x : \tau \vdash e \Rightarrow e' : \tau'}{A \vdash \lambda x_\tau . e \Rightarrow \lambda x_\tau . e' : \tau \rightarrow \tau'}$$

$$\frac{A \vdash e_1 \Rightarrow e_1' : \tau_1 \qquad A \vdash e_2 \Rightarrow e_2' : \tau_2}{A \vdash e_1, e_2 \Rightarrow e_1', e_2' : \tau_1 \times \tau_2}$$

$$A \vdash \underline{\mathrm{nil}}_\tau \Rightarrow \underline{\mathrm{nil}}_\tau : [\tau]$$

$$\frac{A \vdash e \Rightarrow e' : \tau_1 \times \tau_2}{A \vdash e{\downarrow} i \Rightarrow e'{\downarrow} i : \tau_i} \quad i = 1, 2$$

$$\frac{A \vdash e \Rightarrow e' : [\tau]}{A \vdash \underline{\mathrm{tl}}\ e \Rightarrow \underline{\mathrm{tl}}\ e' : [\tau]}$$

$$\frac{A \vdash e_1 \Rightarrow e_1' : \tau_1 \rightarrow \tau_2 \qquad A \vdash e_2 \Rightarrow e_2' : \tau_1}{A \vdash e_1\, e_2 \Rightarrow e_1'\, e_2' : \tau_2}$$

$$\frac{A \vdash e \Rightarrow e' : \tau 1 \qquad \vdash \tau 1 \leq \tau 2 \Rightarrow f}{A \vdash e \Rightarrow f\ e' : \tau 2}$$

Figure 3: Typing Rules

application, one can either coerce the function part to adapt to the argument or *vice versa*. The individual coercions themselves can be carried out in many ways. The important point is that, as a result of the coherence property, this flexibility does not cause any semantic ambiguity.

5 The Semantics in Outline

The semantics of the object language is based on transforming scaled expressions—all expressions are assumed to be scaled—to unscaled ones based on the subtype structure of the last section. The "engine" for the transformation is type inference, specified by a set of typing rules. We use typing rules in which the insertion of coercions is made explicit, departing from previous usage [CW85, Rey85] for systems based on subtypes. One reason is that the statements and proofs of several theorems are made clearer and simpler by the change. We also use the new form to emphasize that our subtype scheme is coercive rather than inclusive. Many recent papers on type inference with subtypes [Car84, Mit88, Tha88] use inclusive subtyping. Coercive subtyping allows relationships that are semantically more *ad hoc*, and need more justification through properties such as coherence. The general form of a typing rule is $A \vdash e \Rightarrow e' : \tau$, which can be read as: "Given a set A of typing assumptions for free variables, the expression e is coerced to e' which has the type τ." The expression e' is the unscaled version of e. The typing rules are given in Figure 3. The most notable rule is the last rule in the right column, which uses a coercion function to account for the use of a subsumption.

The semantics of the coerced expressions derived by type inference is meant to be transparent. This is equivalent to saying that given $A \vdash e \Rightarrow e' : \tau$, the assertion "$e'$ has the type τ" is *prima facie* sound. Suppose there are functions **E** and **T** which map syntactic expressions in the object and type languages to their respective denotations (the details of the definitions of **E** and **T** are standard: see, *e.g.*, [Car88]). The function **E** uses an additional environment argument η as is usual in denotational semantics. We use $\eta \models A$ to mean that the environment η satisfies the type assumptions in A. Note that **E** only assigns *transparent* meanings (without any attempt to resolve scaling) and is only meant to be applied to unscaled expressions.

Semantic Soundness Theorem. $A \vdash e \Rightarrow e' : \tau$ implies $\forall \eta \models A.\ \mathbf{E}[\![e']\!]\eta \in \mathbf{T}[\![\tau]\!]$.

Given that $\mathbf{E}[\![f]\!]\eta \in \mathbf{T}[\![\tau 1 \to \tau 2]\!]$ for the coercion f in the subtyping rule, the proof of this theorem is easy by induction on the stucture of e, and is left as an exercise. Although typing is sound, it is highly nondeterministic. Suppose we define:

$$\mathsf{Type}(A,e) = \{\tau \mid A \vdash e \Rightarrow e' : \tau \text{ for some } e'\} \qquad \mathsf{Expr}(A,e,\tau) = \{e' \mid A \vdash e \Rightarrow e' : \tau\}$$

Type(A,e) is not a singleton for most well-typed e, and Expr(A,e,τ) is not a singleton for most types τ in Type(A,e). However, as we describe in Sections 6 and 7, (*Typexprs*, $\leq$) is a poset and each nonempty Type(A,e) contains a minimal element, which we denote by MinType(A,e). Moreover, although Expr(A,e,τ) may contain many expressions, the semantic coherence theorem in Section 7 asserts that this is semantically inconsequential since all members of Expr(A,e,τ) always have the same (transparent) meaning. To be more precise, the semantic coherence theorem asserts that for all distinct e_1 and e_2 in any Expr(A,e,τ), $\forall \eta \models A$, $\mathbf{E}[\![e_1]\!]\eta = \mathbf{E}[\![e_2]\!]\eta$, and therefore **E** can be applied to Expr(A,e,τ). We can now define the new semantic function **SE** which gives meaning to *scaled* expressions directly. Assuming $\eta \models A$:

$$\mathbf{SE}[\![e]\!]\eta \;=\; \textbf{if } \mathsf{Type}(A,e) = \varnothing \textbf{ then } \mathsf{wrong} \textbf{ else } \mathbf{E}[\![\mathsf{Expr}(A,e,\mathsf{MinType}(A,e))]\!]\eta$$

where wrong is a special semantic value that denotes type error.

6 Properties of the Subtype Structure

In this section we discuss the two major properties of our subtype structure which are needed to validate the semantics outlined in the last section, namely, partial ordering and semantic coherence. The former is needed for the existence of minimal types and the latter for the coherence of typing judgements. A subtype relation is naturally reflexive and transitive (a preorder), as reflected in rules RFLX and TRNS in Figure 2. We begin by showing that (*Typexprs*, $\leq$) is antisymmetric as well. We then outline a proof of the coherence of subtyping judgements, *i.e.*, the property that $\vdash \tau 1 \leq \tau 2 \Rightarrow f$ and $\vdash \tau 1 \leq \tau 2 \Rightarrow g$ implies $f = g$ (extensionally). The property is needed because each subsumption $\tau_1 \leq \tau_2$ can usually be derived in a number of different ways, leading to superficially different coercion functions. For instance:

$$\vdash\ [\mathsf{int} \times \mathsf{int}] \to \mathsf{int} \;\leq\; [[\mathsf{int}] \times [\mathsf{int}]] \to [\mathsf{int}] \;\Rightarrow\; (\lambda g.\, g \bullet (\alpha\ \mathsf{trans})) \bullet \alpha$$

$$\vdash\ [\mathsf{int} \times \mathsf{int}] \to \mathsf{int} \;\leq\; [[\mathsf{int}] \times [\mathsf{int}]] \to [\mathsf{int}] \;\Rightarrow\; \alpha \bullet (\lambda g.\, g \bullet \mathsf{trans})$$

via two derivations for the same subsumption. The details are left as an exercise.

The key to the entire analysis in this section is an analogy between derivations of subtyping judgements and term rewriting sequences. There is room here only to sketch the development. The new technical notion underlying the analogy is that of a *unit* subsumption, corresponding to a single rewriting step. The derivation of a unit subsumption involves exactly one use of one of the *basic* rules (SCL, ZIP, REPL, REPR), along with possible uses of other (nonbasic) rules.

We use the notation $\vdash \tau \lhd \sigma \Rightarrow f$ for unit subsumptions, or just $\tau \lhd \sigma$ for short, whenever we can ignore the coercion *f*. It is not hard to see that the coercion for a unit subsumption is unique. Moreover, any derivation of a subtyping judgement ($\tau_1 \leq \tau_2$) can be presented in the form of a sequence $\tau_1 = \tau_{11} \lhd \ldots \lhd \tau_{1n} = \tau_2$, $n \geq 0$, where the overall coercion is the composition of the unit coercions. The proof of this observation requires a rearrangement of the derivation along the lines of the (quite different) rewriting system described in [CG89]. This constitutes a *partial* normalization of the derivation of the subtyping judgement: there are in general many such sequences for a given judgement. The main result we wish to prove is that all sequences of unit subsumptions for a given subtyping judgement are semantically equivalent, *i.e.*, there is a *unique* representative sequence which represents the subsumption semantically (in terms of the implied coercion). This amounts to a full normalization result for derivations of subtyping judgements.

Note that each step $\tau_i \lhd \tau_{i+1}$ involves replacement of a single subexpression within τ_i by the corresponding expression according to the *basic* rule involved. This is very similar to a term rewriting step and would be just ordinary rewriting based on a set of first-order rewrite rules if not for the antimonotonicity of "$\rightarrow$" in its first argument. To make the analogy more precise, we need to partition occurrences of subexpressions in type expressions into positive and negative ones in order to indicate whether they are monotonically or antimonotonically related to the overall expression. An *occurrence* is a *binary* string specifying a *path* to the subexpression concerned. The subexpression reached by (occurring at) p in τ is denoted by τ/p. The idea is the same as in rewriting, with type constructors and constants playing the role of function symbols. The concatenation of occurrences p and q is denoted by $p \cdot q$. The set of all occurrences in an expression τ will be denoted by $O(\tau)$. The root occurrence Λ is positive. There are four inductive cases for extensions of each $p \in O(\tau)$.

1. $\tau/p = \iota$: there are no occurrences extending p.
2. $\tau/p = [\tau']$: $p \cdot 0$ has the same sign as p.
3. $\tau/p = \tau_1 \times \tau_2$: $p \cdot 0$ and $p \cdot 1$ have the same sign as p.
4. $\tau/p = \tau_1 \rightarrow \tau_2$: $p \cdot 0$ has the opposite and $p \cdot 1$ has the same sign as p.

We wish to think of the basic rules SCL, ZIP, REPL, and REPR as rewrite rules, except that they may be used in either direction depending on the sign of the occurrence being replaced. A "redex" will be either a positive occurrence of an instance of a LHS or a negative occurrence of an instance of a RHS of a basic rule. The corresponding reducts will be the corresponding instances of the RHS and LHS respectively. It is easy to show that τ_1 can be "rewritten" to τ_2 in one step according to this description *iff* $\tau_1 \lhd \tau_2$. Whenever we wish to emphasize the occurrence p involved in a step $\tau_1 \lhd \tau_2$, we shall write it as $\tau_1 \lhd_p \tau_2$.

6.1 Antisymmetry

To prove that $\leq$ is antisymmetric, we define a linearly ordered "measure" for types which strictly grows with $\lhd$. The measure uses the auxiliary functions **D**, F and S. Of these, **D** will play a central role throughout this and the next section. $\mathbf{D}(\tau)$ can be thought of as the depth of (list) structure in τ. We define the measure here and leave the proof to the reader for lack of space.

$$
\begin{array}{llcll}
\mathbf{D}(\tau) = & \text{Case } \tau \text{ of} & & & \\
\iota : & 0 & \mid & \tau_1 \rightarrow \tau_2 : & 0 \\
[\tau'] : & 1 + \mathbf{D}(\tau') & \mid & \tau_1 \times \tau_2 : & \max(\mathbf{D}(\tau_1), \mathbf{D}(\tau_2))
\end{array}
$$

One of the useful properties of **D** is that $\tau_1 \leq \tau_2$ implies $\mathbf{D}(\tau_1) = \mathbf{D}(\tau_2)$. Suppose $<k_1, \ldots, k_n>$ denotes the (lexicographically ordered) sequence of integers k_i, $1 \leq i \leq n$, $P_i(\tau)$ is the number of *positive* occurrences and $N_i(\tau)$ is the number of *negative* occurrences of length i and form $[\tau']$ in τ, and $Depth(\tau)$ is the length of the longest occurrence in $O(\tau)$. Let $\chi_\tau(p)$ denote -1 if p is a negative occurrence in $O(\tau)$ and 1 otherwise.

$$F(\tau) = <k_0, \ldots, k_{Depth(\tau)}> \qquad \text{where } k_i = P_i(\tau) - N_i(\tau),\ 0 \leq i \leq Depth(\tau)$$

$$S(\tau) = \sum_{p \in O(\tau)} \chi_\tau(p) * s_\tau(p) \qquad s_\tau(p) = \begin{cases} \mathbf{D}(\tau 1) + \mathbf{D}(\tau 2), & \tau/p = \tau 1 \to \tau 2 \\ 0, & \text{otherwise} \end{cases}$$

The required measure is the lexicographically ordered pair $(F(\tau), S(\tau))$.

6.2 Coherence of Subsumptions

The key idea in proving coherence of subsumptions is that of *permutations* of sequences of unit subsumptions. The idea is again taken from work on term rewriting [HL79]. A permutation of a sequence is a reordering of the steps in it, preserving the end points. All permutations of a given sequence constitute a *permutation class.* The first step in the proof of coherence of subsumptions is to show that all sequences in the same permutation class are semantically equivalent, given that the basic coercions obey a set of algebraic laws. The second step shows that all sequences for a given subtyping judgement belong to a single permutation class; in other words, there is a unique sequence for each subsumption *modulo* permutations.

Suppose we identify sequences of unit subsumptions by names. Let $B, C, \ldots$ range over sequences. We shall write $B : \tau_1 \leq \tau_2$ to indicate that B is a sequence for $\tau_1 \leq \tau_2$. Clearly, there is a *unique* coercion from τ_1 to τ_2 associated with a given *sequence* $B : \tau_1 \leq \tau_2$. This coercion will be denoted by $\mathbf{C}_B$. Coherence of subsumptions can now be paraphrased as the

Unique Coercion Theorem. $B : \tau_1 \leq \tau_2$ and $C : \tau_1 \leq \tau_2$ implies $\mathbf{C}_B = \mathbf{C}_C$.

Permutations can be defined by using an idea analogous to the classical notion of *residuals* in rewriting [HL79]. It is not hard to show that each redex occurrence except p in τ_1 leaves exactly one residual occurrence in τ_2 when $\tau_1 \lhd_p \tau_2$. Moreover, the residual of a redex is a redex. If $q \neq p$ is such a redex occurrence in τ_1, then let $q \backslash p$ denote its residual in τ_2. Similarly, let $p \backslash q$ denote the residual of p after the alternative step $\tau_1 \lhd_q \tau_3$ which is obviously possible as well. The basic fact we are interested in is that in this situation there is always a τ_4 such that both B: $\tau_1 \lhd_p \tau_2 \lhd_{q \backslash p} \tau_4$ and C: $\tau_1 \lhd_q \tau_3 \lhd_{p \backslash q} \tau_4$ are possible. Note that this does *not* imply that the rewrite relation ($\lhd$) is strongly locally confluent since it assumes that p and q are *distinct.* We shall say that B and C are *direct* permutations of each other, denoted by $B \approx C$. Also, if $B' : \tau_0 \leq \tau_1$ and $C' : \tau_4 \leq \tau_5$ are any other sequences, then $B' \cdot B \cdot C' \approx B' \cdot C \cdot C'$ where $B \cdot C$ denotes the concatenation of sequences B and C.

The general permutation relation, which is the reflexive, transitive and symmetric closure of $\approx$, will be denoted by "$\equiv$". A permutation class is just an equivalence class of "$\equiv$". The justification for using the equivalence notation is that permutations are *semantically* equivalent, *i.e.*, $B \equiv C$ implies $\mathbf{C}_B = \mathbf{C}_C$. To show this we need only prove that the sequences B and C used in defining "$\approx$" correspond to the same coercion. When neither of the two occurrences p and q is a prefix of the other, the two coercions are obviously independent. Suppose one *is* a prefix of the other. There are four cases depending on which of the four *basic* rules (SCL, ZIP, REPL, REPR) is applicable to the *larger* of the two subexpressions (reached by the prefix occurrence). Suppose SCL is applicable, and the smaller subexpression occurs in the argument part of the type. For instance, suppose $f : \tau 1 \to \tau 2$, and there is a type $\tau 3 \lhd \tau 1$, with the corresponding direct coercion g. We have the two sequences

1. $\tau 1 \to \tau 2 \lhd [\tau 1] \to [\tau 2] \lhd [\tau 3] \to [\tau 2]$ 2. $\tau 1 \to \tau 2 \lhd \tau 3 \to \tau 2 \lhd [\tau 3] \to [\tau 2]$

We must show that the equation

$$\text{(SCL)} \quad (\alpha f) \bullet (\alpha \ g) = \alpha \ (f \bullet g)$$

for the corresponding coercion functions holds irrespective of the values of f and g. This is easy to verify—the equation is given in [Bac78] as equation III.4. It is also easy to see that this equation implies the equality of the two coercions derived in the example at the beginning of Section 6. The same equation suffices (with f and g reversing roles) if the smaller subexpression occurs in the result part ($\tau 2$). The other cases require verification of similar simple equations. We list the equations corresponding to ZIP, REPL and REPR below, and leave their derivation and verification to the reader.

$$\text{(ZIP)} \quad (\alpha \ \{f \bullet \mathbf{1}, g \bullet \mathbf{2}\}) \bullet \text{trans} = \text{trans} \bullet \{(\alpha \ f) \bullet \mathbf{1}, (\alpha \ g) \bullet \mathbf{2}\}$$
$$\text{(REPL)} \quad \text{distl} \bullet \{\mathbf{1}, \alpha f \bullet \mathbf{2}\} = (\alpha \ \{\mathbf{1}, f \bullet \mathbf{2}\}) \bullet \text{distl}$$
$$\text{(REPR)} \quad \text{distr} \bullet \{\alpha f \bullet \mathbf{1}, \mathbf{2}\} = (\alpha \ \{f \bullet \mathbf{1}, \mathbf{2}\}) \bullet \text{distr}$$

These equations imply the semantic equivalence of permutations:

Lemma 1: $B \equiv C$ implies $\mathbf{C}_B = \mathbf{C}_C$.

To prove that all sequences for a given subsumption $\tau 1 \leq \tau 2$ belong to the same permutation class is not hard but is technically rather complicated. The main idea is that any sequence can be permuted to a standard form. This is possible because any sequence consists conceptually of a number of subsequences, each corresponding to a single step of scaling. For instance, the algorithm Φ defined in the next section gathers subsequences for zipping and replication together whenever possible. That is, if the set $S = \{ [\tau'] \mid \tau 1 \times \tau 2 \leq [\tau'] \}$ is nonempty, then $\Phi(\tau 1 \times \tau 2)$ is its *least* member, otherwise $\Phi(\tau 1 \times \tau 2)$ fails. Similar properties apply to subsequences for scaling. This leads to:

Lemma 2: $B : \tau_1 \leq \tau_2$ and $C : \tau_1 \leq \tau_2$ implies $B \equiv C$.

The Unique Coercion Theorem is a direct consequence of Lemmas 1 *and* 2.

7 Typing Algorithms

The main result in this section is a minimal typing algorithm for the typing system of Sections 4 and 5. More precisely, we give an algorithm **Type** which, given a set of type assumptions and an expression, will return a coerced expression and its type, and will satisfy the following three properties.

Correctness. If **Type**(A,e) succeeds and returns e', τ then $A \vdash e \Rightarrow e' : \tau$.

Minimality of typing. If $\text{Type}(A,e) \neq \varnothing$ then **Type**(A,e) succeeds and returns $(e', \text{MinType}(A,e))$ (for some e').

Suppose we define the relation $(\equiv_A)$ of "semantic equivalence *modulo* a set A of typing assumptions" by: $e_1 \equiv_A e_2 \Leftrightarrow \forall \eta \models A.\ \mathbf{E}[\![e_1]\!]\eta = \mathbf{E}[\![e_2]\!]\eta$.

Minimality of coercion. If $\tau \in$ Type(A,e), $e' \in$ Expr(A,e,τ), **Type**$(A,e) = e_0, \tau_0$, and $\vdash \tau_0 \leq \tau .\Rightarrow f$, then $e' \equiv_A (f\ e_0)$.

Minimality of coercion asserts that **Type** not only finds a minimal type but also a minimal coerced version in a precise sense. It is easy to see that this implies coherence of typing, *i.e.*, the property that $A \vdash e \Rightarrow e_1 : \tau$ and $A \vdash e \Rightarrow e_2 : \tau$ implies $e_1 \equiv_A e_2$.

Not surprisingly, the interesting part of type inference in our system is the inference of subsumptions. The subtyping rules of Section 4 are complex enough to make this nontrivial. Reynolds [Rey85] points out that minimal typing for sufficiently rich languages—those with "cons" operators or conditional expressions for instance—actually requires inference of least upper bounds (LUBs) and greatest lower bounds (GLBs) for pairs of types which have upper and lower bounds respectively. Subsumption is a special case where the LUB of two types is equal to one of them. We therefore begin with the (mutually recursive) algorithms **LUB** and **GLB**, and then give the minimal typing/coercion algorithm **Type**.

The basic idea in finding the LUB of types $\tau 1$ and $\tau 2$ is to coerce them both to the same outward form with as little change as possible, and then apply the idea recursively to their parts. When one is a product and the other a list type, the product type must be coerced to a list type to achieve compatibility. This is done by the algorithm Φ given below.

```
$\Delta(\tau) =$ if $\tau = \tau' \times \tau''$ then return $\Phi(\tau)$
        else if $\tau = [\tau']$ then return $\tau$ else fail

$\Phi(\tau_1 \times \tau_2) =$ if $\tau_1 = \tau_2 = \iota$ then fail
        else if $\tau_1 \neq \iota$ then let $[\tau_1'] = \Delta(\tau_1)$ else let $\tau_1' = \tau_1$
            if $\tau_2 \neq \iota$ then let $[\tau_2'] = \Delta(\tau_2)$ else let $\tau_2' = \tau_2$
            return $[\tau_1' \times \tau_2']$
```

Example: $\Phi([\text{int} \to \text{int}] \times ([\text{int}] \times \text{int})) = [(\text{int} \to \text{int}) \times (\text{int} \times \text{int})]$

It is easy to see that $\tau \leq \Phi(\tau)$ whenever $\Phi(\tau)$ succeeds. If the set $S = \{ [\tau'] \mid \tau 1 \times \tau 2 \leq [\tau'] \}$ is nonempty, then $\Phi(\tau 1 \times \tau 2)$ is its *least* member, otherwise $\Phi(\tau 1 \times \tau 2)$ fails. Likewise, if $S = \{ [\tau'] \mid \tau \leq [\tau'] \}$ is nonempty, then $\Delta(\tau)$ is its *least* member, otherwise $\Delta(\tau)$ fails.

The GLB algorithm needs a similar function Γ with properties which are the reverse of Φ — it requires a list type to be "uncoerced" to a product type by a *reverse* subsequence. This is a little tricky since given a list type $\tau 2$, the "closest" product type $\tau 1$ for the required minimal subsumption $\tau 1 \leq \tau 2$ is not unique. It is therefore necessary to provide Γ with *both* the (product and list) types for which a GLB is required, so that it can find a starting point for the sequence which is compatible with the given product type. Let $[\tau]^k$ denote the k-fold application of the list constructor to τ. The product and list types are the first and second arguments of Γ:

```
$\Gamma(\tau_{11} \times \tau_{12}, \tau_2) =$ if $\tau_2 \neq [\tau_{21} \times \tau_{22}]^k$ then fail
        else let $k_1 = \mathbf{D}(\tau_{11}) - \mathbf{D}(\tau_{21})$ and $k_2 = \mathbf{D}(\tau_{12}) - \mathbf{D}(\tau_{22})$
            if ($k_1 < k$ and $\tau_{21} \neq \iota$) or ($k_2 < k$ and $\tau_{22} \neq \iota$) then fail
            else if ($k_1 \neq k$ and $k_2 \neq k$) or $k_1 > k$ or $k_2 > k$ then fail
            else return $[\tau_{21}]^{k_1} \times [\tau_{22}]^{k_2}$
```

Example: Suppose $\tau = [\text{int} \times ([\text{int}] \to [\text{int}])]$.

$\Gamma(\text{int} \times ([[\text{int}]] \to [[\text{int}]]), \tau) = \text{int} \times [[\text{int}] \to [\text{int}]]$ $\quad \Gamma([\text{int}] \times [[\text{int}] \to [\text{int}]], \tau) = [\text{int}] \times [[\text{int}] \to [\text{int}]]$

```
LUB $(\tau_1, \tau_2)$ = Case $\tau_1$ of
$\iota$:                 if $\tau_2 = \iota$ then return $\iota$ else fail
$\tau_{11} \to \tau_{12}$:   if $\tau_2 \neq \tau_{21} \to \tau_{22}$ then fail
                    else if not $(k = D(\tau_{11}) - D(\tau_{21}) = D(\tau_{12} - D(\tau_{22}))$ for some $k$ then fail
                    else if  $k = 0$: return $GLB(\tau_{11}, \tau_{21}) \to LUB(\tau_{12}, \tau_{22})$
                             $k > 0$: return $LUB(\tau_1, [\tau_{21}]^k \to [\tau_{22}]^k)$
                             $k < 0$: return $LUB([\tau_{11}]^k \to [\tau_{12}]^k, \tau_2)$
$\tau_{11} \times \tau_{12}$:   if $\tau_2 = [\tau_2']$ then return $LUB(\Phi(\tau_1), \tau_2)$
                   else if $\tau_2 \neq \tau_{21} \times \tau_{22}$ then fail
                   else if $D(\tau_{11}) \neq D(\tau_{21})$ or $D(\tau_{12}) \neq D(\tau_{22})$ then return $LUB(\Phi(\tau_1), \Phi(\tau_2))$
                    else return $LUB(\tau_{11}, \tau_{21}) \times LUB(\tau_{12}, \tau_{22})$
$[\tau_1']$:              if $\tau_2 = [\tau_2']$ then return $[LUB(\tau_1', \tau_2')]$
                    else if $\tau_2 \neq \tau_{21} \times \tau_{22}$ then fail
                    else return $LUB(\tau_1, \Phi(\tau_2))$

GLB $(\tau_1, \tau_2)$ = Case $\tau_1$ of
$\iota$:                 if $\tau_2 = \iota$ then return $\iota$ else fail
$\tau_{11} \to \tau_{12}$:   if $\tau_2 \neq \tau_{21} \to \tau_{22}$ then fail
                    else if not $(k = D(\tau_{11}) - D(\tau_{21}) = D(\tau_{12} - D(\tau_{22}))$ for some $k$ then fail
                    else if  $k = 0$: return $LUB(\tau_{11}, \tau_{21}) \to GLB(\tau_{12}, \tau_{22})$
                             $k > 0$: return $GLB([\tau_{11}]^{-k} \to [\tau_{12}]^{-k}, \tau_2)$
                             $k < 0$: return $GLB(\tau_1, [\tau_{21}]^k \to [\tau_{22}]^k)$
$[\tau_1']$:              if $\tau_2 = [\tau_2']$ then return $[GLB(\tau_1', \tau_2')]$
                    else if $\tau_2 \neq \tau_{21} \times \tau_{22}$ then fail
                    else return $GLB(\Gamma(\tau_2, \tau_1), \tau_2)$
$\tau_{11} \times \tau_{12}$:    if $\tau_2 = \tau_{21} \times \tau_{22}$ then return $GLB(\tau_{11}, \tau_{21}) \times GLB(\tau_{12}, \tau_{22})$
                    else if $\tau_2 = [\tau_2']$ then return $GLB(\tau_1, \Gamma(\tau_1, \tau_2))$
                    else fail
```

Figure 5: LUB and GLB Algorithms

Note the extensive use of the "depth of list structure" function **D** in both Γ and in the **LUB/GLB** algorithm. There is no room here for a complete explanation of its role, but the function is used to measure the mismatch in degree of scaling and/or zipping/replication between type expressions. The **LUB, GLB** algorithms are given in Figure 5. Given Δ, **LUB** and **GLB**, the algorithm **Type** for inference of minimal types and coercions—given in Figure 6—is straightforward except for the application case, which needs to resolve the overloading of the

```
$\mathbf{Type}(A, e)$ = Case $e$ of
  $x$: return $x, Ax$ ;   $\underline{\text{nil}}_\tau$: return $\underline{\text{nil}}_\tau$, $[\tau]$
  $e_1, e_2$: let $e_3, \tau_3 = \mathbf{Type}(A, e_1)$ and $e_4, \tau_4 = \mathbf{Type}(A, e_2)$ in return $(e_3, e_4), \tau_3 \times \tau_4$
  $e \downarrow i$: if $\mathbf{Type}(A, e) \neq e', \tau_1 \times \tau_2$ then fail else return $e' \downarrow i, \tau_i$  $(i = 1$ or $2)$
  $e_1; e_2$ : let $e_3, \tau_3 = \mathbf{Type}(A, e_1)$ and $e_4, \tau_4 = \mathbf{Type}(A, e_2)$ in
          let $\tau 5 = \mathbf{LUB}(\tau_3, \tau_4)$ in return $(C_{\tau_3 \leq \tau_5}\ e_3, C_{\tau_4 \leq \tau_5}\ e_4), [\tau_5]$
  $\underline{\text{hd}}\ e$: let $e', \tau = \mathbf{Type}(A, e)$ in if $\Delta(\tau)$ returns $[\tau']$ then return $\underline{\text{hd}}\ (C_{\tau \leq [\tau']}\ e'), \tau'$ else fail
  $\underline{\text{tl}}\ e$: let $e', \tau = \mathbf{Type}(A, e)$ in if $\Delta(\tau)$ returns $[\tau']$ then return $\underline{\text{tl}}\ (C_{\tau \leq [\tau']}\ e'), [\tau']$ else fail
  $\lambda x_\tau . e$ : let $e', \tau' = \mathbf{Type}(A{+}x{:}\tau, e)$ in return $\lambda x_\tau . e', \tau \rightarrow \tau'$
  $\underline{\text{fix}}\ e$: if $\mathbf{Type}(A, e) \neq e', \tau_1 \rightarrow \tau_2$ for some $e', \tau_1, \tau_2$ then fail
              else if $\mathbf{LUB}(\tau_1, \tau_2) \neq \tau_1$ then fail else return $e' \bullet C_{\tau_2 \leq \tau_1}, \tau_2$
  $e_1\ e_2$: if $\mathbf{Type}(A, e_1) \neq e_1', \tau_1 \rightarrow \tau_2$ for some $e_1', \tau_1, \tau_2$ then fail
            else let $e_2', \tau_3 = \mathbf{Type}(A, e_2)$ and $k = \mathbf{D}(\tau_3) - \mathbf{D}(\tau_1)$ in
              if $k < 0$ or $\mathbf{LUB}([\tau_1]^k, \tau_3) \neq [\tau_1]^k$
                  then fail else return $(\alpha^k\ e_1')\ (C_{\tau_3 \leq [\tau_1]^k}\ e_2'), [\tau_2]^k$
```

Figure 6: Algorithm Type

function part—the constant k derived in the analysis of this case (using **D** again) captures the only meaning of the function part that could possibly be appropriate in that application.

8 Concluding Remarks

We have described a coercion based semantics for implicit scaling which rests on just four structural subtyping rules. The rest of the system simply works out the consequences of applying these rules within a standard general framework of the kind described in [Rey85]. We regard this as a nice illustration of the way coercive structural subtyping can be used—at little cost in semantic complexity—to raise the expressive power of a language by eliminating a class of programming chores. For other applications of the idea, see [BC+89, Tha90].

The compatibility of the subtype structure described here with parametric polymorphism is an interesting topic for further investigation. Combining our system with the *implicit* parametric polymorphism of the Hindley-Milner system [DM82] may result in the loss of semantic coherence. Consider for instance our operator $\underline{\text{hd}}$. Under parametric polymorphism, $\underline{\text{hd}}$ is usually a polymorphic *function* which possesses all types of the form $[\tau] \rightarrow \tau$. Assuming loosely that the set A of type assumptions can supply any of the types possessed by such a function $\underline{\text{hd}}$, we would have

$$A \vdash \underline{\text{hd}} \Rightarrow \underline{\text{hd}} : [[\text{int}]] \rightarrow [\text{int}], \quad \text{since } A(\underline{\text{hd}}) = [[\text{int}]] \rightarrow [\text{int}]$$
$$A \vdash \underline{\text{hd}} \Rightarrow \alpha\, \underline{\text{hd}} : [[\text{int}]] \rightarrow [\text{int}], \quad \text{since } A(\underline{\text{hd}}) = [\text{int}] \rightarrow \text{int} \leq [[\text{int}]] \rightarrow [\text{int}]$$

where the two converted expressions obviously have different meanings in most applications.

One way to overcome this difficulty is to avoid treating the operations on the "structure of interest" as functions. This may be more natural in some cases (*e.g.*, arrays) than in others (*e.g.*, lists). It might be possible to avoid this dilemma in a combination with *explicit* bounded abstraction over types [CW85]. Besides generalizing our system, a successful combination with the latter would provide some insight into general techniques for combining coercive subtyping with parametric polymorphism. A similar combination has been studied in [BC+89] for a coercive interpretation of structural subtyping for labeled records.

References

[Bac78] Backus, J. Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. CACM **21**, 613-641 (1978)

[Ben85] Benkard, J. P. Control of Structure and Evaluation. In: Proc. APL'85 Conference.

[BC+89] Breazu-Tannen, V., Coquand, T., Gunter, C. and Scedrov, A. Inheritance and explicit coercion. In: Proceedings of Fourth LICS Symposium. IEEE 1989

[Bre88] Breuel, T.M. Data Level Parallel Programming in C++. In: Proc. of 1988 USENIX C++ Conf., pp. 153-167.

[Car88] Cardelli, L. A Semantics of Multiple Inheritance. Info. and Comp. **76**,138-164 (1988)

[CG89] Curien, P-L. and Ghelli, G. Coherence of Subsumption. Manuscript. 1989

[CW85] Cardelli, L. and Wegner, P. On Understanding Types, Data Abstraction and Polymorphism. Computing Surveys **17** (4) (1985)

[DM82] Damas, L. and Milner, R. Principle Type-schemes for Functional Programs. In: Proc. 9th POPL Symposium, Albuquerque, NM. ACM 1982

[HL79] Huet, G. and Levy, J-J. Computations in Nonambiguous Linear Term Rewriting Systems. Tech. Rep. 359, INRIA-Le Chesney, France, 1979

[HS86] Hillis, W.D. and Steele, G.L., Jr. Data Parallel Algorithms. CACM **29** (12) (1986)

[Ive62] Iverson, K.E. A Programming Language. Wiley, New York, 1962

[JM78] Jenkins, M.A. and Michel, J. Operators in an APL Containing Nested Arrays. CIS Dept., Queen's University, Kingston, Ontario. Tech. Rep. 78-60. 1978

[Mil84] Milner, R. A Proposal for Standard ML. In: Proc. 1984 ACM Symp. on LISP and Functional Programming, Austin, TX., pp. 184-197. ACM 1984

[Mit88] Mitchell, J.C. Polymorphic Type Inference and Containment. Information and Computation **76**, 211-249 (1988)

[Rey80] Reynolds, J.C. Using Category Theory to Design Implicit Conversions and Generic Operators. In: Semantics Directed Compiler Generation (N.D.Jones, Ed.), pp. 211-258, Lecture Notes in Computer Science, Vol. 94. Springer-Verlag 1980

[Rey85] Reynolds, J.C. Three Approaches to Type Structure. In: Proc. TAPSOFT 1985, Lecture Notes in Computer Science, Vol. 186. Springer-Verlag 1985

[Tha88] Thatte, S.R. Type Inference with Partial Types. In: Proc. 15th ICALP. Lecture Notes in Computer Science, Vol. 317, pp. 615-629. Springer-Verlag 1988

[Tha90] Thatte, S.R. Quasi-static Typing. In: Proc. of the 17th POPL Symposium, San Francisco, CA. ACM 1990

[Vis89] Vishnubhotla, P. Data Parallel Programming on Transputer Networks. In: Proc. of 2nd Conf. of North American Transputer Users Group, G.S. Stiles (Ed.). April 1989