Using Kinds To Represent Heterogeneous Collections In A Static Type System (Extended Abstract)

MS-CIS-90-62
LOGIC & COMPUTATION 22

Peter Buneman
University of Pennsylvania
Atsushi Ohori
University of Glasgow

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

August 1990

Using Kinds to Represent Heterogeneous Collections in a Static Type System (Extended Abstract)

Peter Buneman*
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, U.S.A.

peter@cis.upenn.edu
Tel. (215) 898-7703

Atsushi Ohori†
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland

ohori%cs.glasgow.ac.uk@nsfnet-relay.ac.uk
Tel. +44 (0)41 339-8855

Abstract

We consider the problem of representing heterogeneous collections of objects in a typed polymorphic programming language in such a way that common properties of members of a collection, such as having commonly named field with a common type can be expressed in the type system. The use of such collections is widespread in object-oriented and database programming and has so far been achieved in statically typed systems only through the use of a single *dynamic* type, which effectively hides all the structure of a value. In this paper we exploit a system of types and *kinds* (sets of types) to represent dynamic values with some known properties. The type system is shown to be sound and to have a complete type inference algorithm.

1 Introduction

Heterogeneous collections are commonly used in object-oriented programming and in certain database systems. For example, one might retrieve from a database two sets of records

$$S_1 = \{\texttt{[Maker = "Ford", Type = "LTD", Range = 405, MPG = 29]},\ \texttt{[Maker = "Peugeot", Type = "405", MPG = 32]},\ \texttt{[Maker = "Honda", Type = "Accord", MPG = 40]}\}$$

$$S_2 = \{\texttt{[Maker = "Ford", Type = "LTD", Range = 405, MPG = 29]},\ \texttt{[Maker = "Ford", Type = "Mustang", Range = 325]}\}$$

Although neither of these sets is homogeneous, the members of each have certain common properties, and it should be possible to extract **Maker, Type** and **MPG** from all members of S_1 and **Maker, Type** and **Range** from all members of S_2. Moreover we would expect to find the intersection of these sets of fields, **Name** and **Type**, available on $S_1 \cup S_2$ and their union available on $S_1 \cap S_2$. Can we make use of such reasoning in a statically typed language?

The ability to deal with heterogeneous collections is claimed [Str87] as an advantage of object-oriented programming, but it is not clear that in statically typed languages that exploit some form of subsumption

*Supported by research grants NSF IRI86-10617, ARO DAA6-29-84-k-0061 and ONR NOOO-14-88-K-0634
†Supported by a British Royal Society Research Fellowship. On leave from OKI Electric Industry, Co., Japan.

based on subtypes or subclasses [Car88, CW85] the collections really are heterogeneous. If, for example, we have $e : \mathit{Employee}$ and $l : \mathit{list}(\mathit{Person})$ where, $\mathit{Employee} \leq \mathit{Person}$, such languages will allow the formation of the list $\mathit{cons}(e, l)$, but this expression is of type $\mathit{list}(\mathit{Person})$ and therefore $\mathit{head}(\mathit{cons}(e, l))$ is of type Person. By placing e on this list we have somehow lost some of its properties, and the type system does not allow us to recover them.

In Amber [Car86] one can package a value together with its type into a value of type `Dynamic`. We can now build homogeneous collections of such values of type `Dynamic` even though the values from which they are constructed have different types. Properties of members of a collection are recovered by explicitly coercing them to some type. This idea was developed in [ACPP89] and in [Tha88]. In the latter a universal sum type Ω was introduced (similar to `Dynamic`) and a type inference method was developed based on the type ordering $\leq$ induced by the relation $\Omega \leq \tau$ lifted to other type constructors.

These approaches are too coarse in that they do not allow us to reason about unions and intersections in the way we have just advocated. In order to allow such reasoning we shall use a refined notion of dynamic types that we shall call *partial* types. A value of such a type can be thought of as a dynamic value with part of its structure "revealed". Thus an assertion of the form e:$\mathcal{P}$`(<Make:string, Range:int>)` (we shall explain the syntax shortly) indicates that e denotes a dynamic value whose `Make` and `Range` information has been revealed, which means that the operations e`.Make` and e`.Range` are legitimate expressions of type `string` and `int` respectively. Moreover the complete type of the value that was used to form the dynamic value e must contain `Make:string` and `Range:int` fields.

`<Make:string, Range:int>` describes a set of types or *kind*. It is the set of all record types that have `Make:string` and `Range:int` fields. The type `[Make:string, Range:int, MPG:int]` is an example of such a type. The notation $\mathcal{P}$`(<Make:string, Range:int>)` describes the type of a dynamic value whose exact type is some member of the kind `<Make:string, Range:int>`. In order to obtain more properties of this value we must either explicitly coerce it to more specific partial type or project it out to its complete type.

In this paper we shall develop a polymorphic system of types and kinds that allows us to manipulate partially typed collections of dynamic values. During our development, we shall sometimes need to assume the concrete structure of collection types. In such cases we shall give accounts of both sets and lists. However, we hope that the system of partial types can be combined with other forms of collections. To deal with sets which require equality on elements, we single out the subsets of types which have computable equality. This complicates our presentation but allows us to treat both sets and lists in a general way. We shall first give the base calculus as an extension of the simply typed lambda-calculus and obtain the results needed to establish the soundness of the type system. We shall then provide a type inference algorithm that allows us to extend ML-style polymorphism to include the additional features. The approach we shall adopt is based upon a proposal originally suggested by Wand [Wan87] and subsequently developed in [OB88, JM88, Mit90, Rém89, Wan89]. Although we shall make use of an ordering on partial types, this ordering is used to express rules for union and intersection and is not needed to derive the polymorphism in field selection. In particular we do not require a subsumption rule which, as we have noted in [BTBO89] creates some problems for database operations such as equality test and *join* [BJO90, Oho89] that require the complete structure of an object. A similar problem was also observed in [CHC90]. Intersection and union, when applied to sets of dynamic values, similarly require an equality test on the complete value rather than the part that is revealed by some partial type.

2 The Base Calculus for Partial Types

2.1 Terms

The set of terms of the base calculus is given by the following syntax:

$$
\begin{aligned}
M \ ::= \ & (c{:}\tau) \mid x \mid \lambda x{:}\tau.M \mid M(M) \mid [l{=}M,\ldots,l{=}M] \mid M.l \mid \mathbf{modify}(M,l,M) \mid \\
& (\{\}{:}\{\tau\}) \mid \{M\} \mid \mathbf{union}(M,M) \mid \mathbf{reduce}(M,M,M,M) \mid \\
& \mathbf{dynamic}(M{:}\tau) \mid \mathbf{fuse}(M,M) \mid \mathbf{as}(M{:}\tau) \mid \mathbf{coerce}(M{:}\tau)
\end{aligned}
$$

$(c{:}\tau)$ stands for typed constants, $[l{=}M,\ldots,l{=}M]$ is the syntax for labeled records, $M.l$ is field selection, $\mathbf{modify}(M_1,l,M_2)$ is field modification (or update) which creates a new record by modifying the l field of the record M_1 to M_2. $\{\}$, and $\{M\}$ denote respectively the empty collection and singleton collections. We write $\{v,\ldots,v\}$ for some canonical representation for the values of collection types. **union** combines two collections. It is set-theoretic union for sets and *append* for lists. **reduce** is a general elimination operation. For lists, this is defined as

$$
\begin{aligned}
&\mathbf{reduce}(f,op,z,(\{\}{:}\{\tau\})) = z \\
&\mathbf{reduce}(f,op,z,\{v_1,v_2,\ldots,v_n\}) = op(f(v_1),\mathbf{reduce}(f,op,z,\{v_2,\ldots,v_n\}))
\end{aligned}
$$

For example,

$$\texttt{sum} = \lambda \texttt{S:\{int\}.reduce}(\lambda \texttt{x:int.x,plus,0,S})$$

where **plus** is binary addition. In order for this to be a definition for sets, we must assume that *op* is an associative and commutative binary operator, but it is not clear that there is any efficient method of checking this. In what follows we assume nondeterministic choice of an order of the elements in the set. In the case that *op* is associative and commutative, the result does not depend on the choice. $\mathbf{dynamic}(M{:}\tau)$, when evaluated, creates a dynamic value, which is the pair (τ, v) of the type and the value denoted by the term M. Although the dynamic value belongs to the domains of various partial types, this expression has the unique type $\mathcal{P}(\langle\tau\rangle)$, which is the "least partial" type for this dynamic value. **fuse** is a form of equality test generalized to partial types. If the two arguments are the same then it returns the singleton collection of the value otherwise it returns the empty collection. The result provides a useful type information when the arguments are dynamic values. For partial types, we have two coercion functions. One is $\mathbf{as}(M,\tau)$ which returns a singleton collection of itself if the dynamic value denoted by M belongs to the partial type τ; otherwise it retuns the empty collection. The other is $\mathbf{coerce}(M,\tau)$ which returns a singleton collection of the value part of the dynamic value denoted by M if its type is the same as τ; otherwise it returns the empty collection. The collection type is used in **fuse**, **as** and **coerce** to avoid run time exceptions.

2.2 Types, Kinds and Typing Rules

The set of types and eqtypes (i.e. types with computable equality) are given as:

$$
\begin{aligned}
\tau \ &::= \ b \mid b^{eq} \mid [l{:}\tau,\ldots,l{:}\tau] \mid \{\tau\} \mid \tau \rightarrow \tau \mid \mathcal{P}(\kappa) \\
\sigma \ &::= \ b^{eq} \mid [l{:}\sigma,\ldots,l{:}\sigma] \mid \{\sigma\} \mid \mathcal{P}(\epsilon)
\end{aligned}
$$

b stands for base types and b^{eq} for those with computable equality. $[l{:}\tau,\ldots,l{:}\tau]$ is the syntax for record types. $\{\tau\}$ stands for collection types. $\mathcal{P}(\cdots)$ are partial types specified by kinds (κ) or eqkinds (ϵ), which are given by following syntax:

$\kappa ::= \langle\tau\rangle \mid \langle l{:}\tau,\ldots,l{:}\tau\rangle \mid \mathbf{any}$

$\epsilon ::= = \langle\sigma\rangle \mid \langle l{:}\sigma,\ldots,l{:}\sigma\rangle^{eq} \mid \mathbf{eq}$

The following *kinding relation* gives the intended meaning of kinds:

$$\begin{aligned}
\tau &:: \langle\tau\rangle \\
\tau &:: \mathbf{any} \\
\sigma &:: \mathbf{eq} \\
[l_1{:}\tau_1,\ldots,\ l_n{:}\tau_n,\ldots] &:: \langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle \\
\mathcal{P}(\langle[l_1{:}\tau_1,\ldots,\ l_n{:}\tau_n,\ldots]\rangle) &:: \langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle \\
\mathcal{P}(\langle l_1{:}\tau_1,\ \ldots,l_n{:}\tau_n,\ldots\rangle) &:: \langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle \\
\mathcal{P}(\langle l_1{:}\sigma_1,\ldots,\ l_n{:}\sigma_n,\ldots\rangle^{eq}) &:: \langle l_1{:}\sigma_1,\ldots,l_n{:}\sigma_n\rangle \\
[l_1{:}\sigma_1,\ldots,\ l_n{:}\sigma_n,\ldots,l_m{:}\sigma_m] &:: \langle l_1{:}\sigma_1,\ldots,l_n{:}\sigma_n\rangle^{eq} \\
\mathcal{P}(\langle[l_1{:}\sigma_1,\ldots,\ l_n{:}\sigma_n,\ldots,l_m{:}\sigma_m]\rangle) &:: \langle l_1{:}\sigma_1,\ldots,l_n{:}\sigma_n\rangle^{eq} \\
\mathcal{P}(\langle l_1{:}\sigma_1,\ldots,\ l_n{:}\sigma_n,\ldots\rangle^{eq}) &:: \langle l_1{:}\sigma_1,\ldots,l_n{:}\sigma_n\rangle^{eq}
\end{aligned}$$

Kinds are used to define typing rules for elimination operations for dynamic values. It also plays an important role in developing a type inference algorithm in section 3.

Partial types are naturally ordered in terms of the amount of static information:

$$\begin{aligned}
\mathcal{P}(\mathbf{any}) &\leq \mathcal{P}(\kappa) \\
\mathcal{P}(\mathbf{eq}) &\leq \mathcal{P}(\epsilon) \\
\mathcal{P}(\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle) &\leq \mathcal{P}(\langle[l_1{:}\tau'_1,\ldots,\ l{:}\tau'_n,\ldots]\rangle) \text{ if } \tau_i \leq \tau'_i\ (1 \leq i \leq n) \\
\mathcal{P}(\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle) &\leq \mathcal{P}(\langle l_1{:}\tau'_1,\ldots,\ l{:}\tau'_n,\ldots\rangle) \text{ if } \tau_i \leq \tau'_i\ (1 \leq i \leq n) \\
\mathcal{P}(\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle) &\leq \mathcal{P}(\langle l_1{:}\sigma'_1,\ldots,\ l{:}\sigma'_n,\ldots\rangle^{eq}) \text{ if } \tau_i \leq \sigma'_i\ (1 \leq i \leq n) \\
\mathcal{P}(\langle l_1{:}\sigma_1,\ldots,l_n{:}\sigma_n\rangle^{eq}) &\leq \mathcal{P}(\langle[l_1{:}\sigma'_1,\ldots,\ l{:}\sigma'_n,\ldots,l_m{:}\sigma'_m]\rangle) \text{ if } \sigma_i \leq \sigma'_i\ (1 \leq i \leq n) \\
\mathcal{P}(\langle l_1{:}\sigma_1,\ \ldots,l_n{:}\sigma_n\rangle^{eq}) &\leq \mathcal{P}(\langle l_1{:}\sigma'_1,\ldots,\ l{:}\sigma'_n,\ldots\rangle^{eq}) \text{ if } \sigma_i \leq \sigma'_i\ (1 \leq i \leq n)
\end{aligned}$$

The ordering on types is the smallest reflexive relation containing the above relation. The ordering has the following property, which will be needed to develop type checking rules:

Proposition 1 *$\leq$ is a partial order with pairwise bounded joins and meets.* ∎

Using these definitions, we can now define the type system as a proof system for *typings* of the form $\mathcal{A} \rhd M : \tau$ where $\mathcal{A}$ is a function from a finite set of variables to types, called a *type assignment*. We write $\mathcal{A}\{x : \tau\}$ for the function $\mathcal{A}'$ such that $dom(\mathcal{A}') = dom(\mathcal{A}) \cup \{x\}$, $\mathcal{A}'(x) = \tau$ and $\mathcal{A}'(y) = \mathcal{A}(y)$ for all $y \in dom(\mathcal{A}), y \neq x$. Figure 1 shows some of the typing rules. If the collection types are sets then the types τ_1, τ_2 in the rule (union) are restricted to eqtypes σ_1, σ_2. The partiality of partial types is increased by the rule (union), decreased by the rule (fuse) and changed by the rule (as). The complete set of typing rules will be given in the full paper.

The calculus has a static type-checking algorithm.

Proposition 2 *For any pre-term M and $\mathcal{A}$, there is at most one τ such that $\mathcal{A} \rhd M : \tau$. Moreover, there is an algorithm which, given $\mathcal{A}$ and M, computes the unique τ such that $\mathcal{A} \rhd M : \tau$ if one exists; otherwise it reports failure.*

(record) $$\frac{\mathcal{A} \rhd M_i : \tau_i \ \ (1 \leq i \leq n)}{\mathcal{A} \rhd [l_1{=}M_1, \ldots, l_n{=}M_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]}$$

(dot) $$\frac{\mathcal{A} \rhd M : \tau_1}{\mathcal{A} \rhd M.l : \tau_2} \quad \text{if } \tau_1 :: <l : \tau_2>$$

(union) $$\frac{\mathcal{A} \rhd M_1 : \{\tau_1\} \qquad \mathcal{A} \rhd M_2 : \{\tau_2\}}{\mathcal{A} \rhd \mathbf{union}(M_1, M_2) : \{\tau_3\}} \quad \text{if } \tau_3 = \tau_1 \sqcap \tau_2$$

(fuse) $$\frac{\mathcal{A} \rhd M_1 : \sigma_1 \qquad \mathcal{A} \rhd M_2 : \sigma_2}{\mathcal{A} \rhd \mathbf{fuse}(M_1, M_2) : \{\sigma_3\}} \quad \text{if } \sigma_3 = \sigma_1 \sqcup \sigma_2$$

(dynamic) $$\frac{\mathcal{A} \rhd M : \tau}{\mathcal{A} \rhd \mathbf{dynamic}(M) : \mathcal{P}(<\tau>)}$$

(as) $$\frac{\mathcal{A} \rhd M : \mathcal{P}(\kappa)}{\mathcal{A} \rhd \mathbf{as}(M : \mathcal{P}(\kappa')) : \{\mathcal{P}(\kappa')\}}$$

Figure 1: Some of the Typing Rules for the Base Calculus

This is proved by proving the stronger statement that any term has at most one typing derivation. The proof is by induction on the number of applications of typing rules. ∎

The next property guarantees that the introduction of partial types preserved the property of simple types:

Proposition 3 *If M does not contain* **dynamic** *then $\mathcal{A} \rhd M : \tau$ is derivable in our system iff it is derivable in the simply typed lambda calculus with records.* ∎

As we noted earlier, this property is needed for operations that require the complete structures of objects.

2.3 Operational Semantics

Following [ACPP89], we give an operational semantics by defining a set of rules to reduce closed terms to canonical values. To do this we define the set of *canonical values* as:

$$v \ ::= \ (c{:}b) \mid \lambda x{:}\tau.M \mid [l{=}v, \ldots, l{=}v] \mid \{v, \ldots, v\} \mid (\tau, v) \mid \mathtt{wrong}$$

and the set of *canonical eqvalues* as:

$$d \ ::= \ (c{:}b^{eq}) \mid [l{=}d, \ldots, l{=}d] \mid \{d, \ldots, d\} \mid (\sigma, d)$$

We have already explained $\{\ldots\}$ and (τ, v). If the collections are sets then we assume an appropriate equivalence relation on values of the form $\{\ldots\}$ and consider them as equivalence classes. **wrong** is the canonical value of a run time type error. We write $M \Longrightarrow v$ to denote that M is reduced to a canonical value v. We define the *extent* $[\![\kappa]\!]$ of a kind κ as $\tau \in [\![\kappa]\!]$ iff there is some τ' such that $\tau \leq \tau'$ and $\tau' :: \kappa$. Figure 2 shows some of the reduction rules. The complete set of rules will be given in the full paper.

The type system is extended to the canonical values we have just introduced by adding the rules:

$$\frac{M \Longrightarrow [l_1{=}v_1,\ldots,l_n{=}v_n] \text{ or } (\tau,[l_1{=}v_1,\ldots,l_n{=}v_n])}{M.l_i \Longrightarrow v_i \ (1 \le i \le n)}$$

$$\frac{M_1 \Longrightarrow d \qquad M_2 \Longrightarrow d}{\mathbf{fuse}(M_1,M_2) \Longrightarrow \{d\}} \quad \text{if } d \text{ is an eqvalue}$$

$$\frac{M_1 \Longrightarrow d_1 \qquad M_2 \Longrightarrow d_2}{\mathbf{fuse}(M_1,M_2) \Longrightarrow \{\}} \quad \text{if } d_1, d_2 \text{ are eqvalues and } d_1 \neq d_2$$

$$\frac{M \Longrightarrow (\tau,v)}{\mathbf{as}(M{:}\mathcal{P}(\kappa)) \Longrightarrow \{(\tau,v)\}} \quad \text{if } \tau \in [\![\kappa]\!]$$

$$\frac{M \Longrightarrow (\tau_1,v)}{\mathbf{as}(M{:}\mathcal{P}(\kappa)) \Longrightarrow \{\}} \quad \text{if } \tau_1 \notin [\![\kappa]\!]$$

Figure 2: Some of The Reduction Rules

$$\text{(collection)} \quad \frac{\mathcal{A} \rhd v_i : \tau \ (1 \le i \le n)}{\mathcal{A} \rhd \{v_1,\ldots,v_1\} : \{\tau\}}$$

$$\text{(dval)} \quad \frac{\mathcal{A} \rhd v : \tau}{\mathcal{A} \rhd (\tau,v) : \mathcal{P}(\kappa)} \text{ for any } \kappa \text{ such that } \tau \in [\![\kappa]\!]$$

Note that the extended terms in general have multiple typings but **wrong** has none. The following theorem establishes the soundness of the type system.

Theorem 1 *For any term M and any canonical expression v, if $\emptyset \rhd M : \tau$ and $M \Longrightarrow v$ then $\emptyset \rhd v : \tau$.*

The typing is preserved by β-reduction, whose proof is similar to that of the subject-reduction theorem [HS86]. The theorem is then proved by using the definitions of typings for term constructors. ∎

Since **wrong** has no typing, we have:

Corollary 1 *For any term M and any canonical expression v, if $\emptyset \rhd M : \tau$ and $M \Longrightarrow v$ then $v \neq$ **wrong**.*

2.4 Programming Examples

We show how these partial types are used in programming with heterogeneous sets. We use pairs (M,N), $M.1$, $M.2$, product type $\tau_1{*}\tau_2$ and n-argument function definition of the form **fun** f $(x,\ldots,x)$ = $\cdots$. Then are easily defined using records.

A function to fuse a dynamic value with a member of a set of dynamic values can be implemented as

fun fuse1(x:σ_1,S:{σ_2}) =
    reduce(λy:σ_2.**fuse**(x,y),λx:{σ_1}*{σ_2}.**union**(x.1,x.2),({}:{$\sigma_1 \sqcup \sigma_2$}),S)

and the intersection of two sets is then implemented as

fun intersect (S1:{σ_1},S2:{σ_2}) =
    reduce(λy:σ_1.**fuse1**(y,S2),λx:{σ_1}*{σ_2}.**union**(x.1,x.2),({}:{$\sigma_1 \sqcup \sigma_2$}),S1)

on partial types, which has the type $\{\sigma_1\} * \{\sigma_2\} \to \{\sigma_1 \sqcup \sigma_2\}$ as expected. In a similar fashion it is straightforward to implement other set operations such as subtraction for sets of dynamic values.

In databases there is an ambiguity about the meaning of the *is-a* relationship. By asserting *Employee is-a Person* we may mean that *Employee* is a subclass of *Person* in the sense that the methods of *Person* are also available for instances of *Employee*. We may also mean that *Employee* and *Person* are sets and the *is-a* relation specifies set inclusion. Notice that these two meanings are incompatible if we assume the sets are uniformly typed, for how can a set of one type be a subset of a set of another type?. However, if we allow sets to be heterogeneous and agree that the *is-a* ordering is (the inverse of) our ordering on types, then we can derive the ordering on sets from the ordering on partial types.

Assume that *Person* is shorthand for the type $\mathcal{P}$(<Name:string, Address:string>) and *Employee* is shorthand for $\mathcal{P}$(<Name:string, Address:string, Salary:int>). The function

fun Persons_of(S:$\mathcal{P}$(any)) =
  reduce(λx:$\mathcal{P}$(any).as(x:*Person*),λx:{*Person*}*{*Person*}.**union**(x.1,x.2),({}:*Person*),S)

is of type {$\mathcal{P}$(any)} $\rightarrow$ {*Person*}

fun Employees_of(S:$\mathcal{P}$(any)) =
  reduce(λx:$\mathcal{P}$(any).as(x:*Employee*),
    λx:{*Employee*}*{*Employee*}.**union**(x.1,x.2),({}:{*Employee*}),S)

is of type {$\mathcal{P}$(any)} $\rightarrow$ {*Employee*} and the semantics of our operations for partial types guarantees us that Employees_of(S) $\subseteq$ Persons_of(S). Thus the inclusion relationship is derived from the relationship on kinds. Moreover, we shall show in the next section that the desired features of *method inheritance* between those sets are automatically achieved by ML style *polymorphism* through *type inference*. To our knowledge no other statically typed programming language allows this coupling of inheritance and subsets.

3 Type Inference Algorithm

Type inference is a method used to infer the set of all derivable typings for a given untyped term. Let e stand for the set of untyped terms obtained form the terms of the base calculus by erasing all type specifications of the form $\lambda x{:}\tau.M'$, ({}:{τ}), **dynamic**($M{:}\tau$). We write $\mathcal{A} \rhd e : \tau$ if there is some M such that e is the type erasure of M and $\mathcal{A} \rhd M : \tau$. The type inference problem is then stated as the problem to construct a representation of the set $\{(\mathcal{A},\tau)|\mathcal{A} \rhd e : \tau\}$

To solve this problem, we define *conditional typing schemes* [OB88] to represent sets of typings. This is the refinement of *type schemes* [Hin69, Mil78] with syntactic conditions on substitutions of type variables, which are needed to represent constraints associated with some of our typing rules. Some of the conditions can be integrated directly in type schemes by refining them to *kinded (eq)type schemes* by introducing kind constraints on (eq)type variables [Oho90]. The appropriate set of kinded type schemes (ranged over by T) and kinded eqtype schemes (ranged over by E) are give as:

$T ::= t^K \mid u^K \mid b \mid b^{eq} \mid [l{:}T,\ldots,l{:}T] \mid T \rightarrow T \mid \mathcal{P}(\kappa)$
$E ::= u^K \mid b^{eq} \mid [l{:}E,\ldots,l{:}E] \mid \mathcal{P}(\epsilon)$

t^K, u^K are *kinded type variables* and *kinded eqtype variables* respectively ranging over types and eqtypes. K denote *kind schemes* which constrain their instantiation. κ, ϵ are kinds and eqkinds we have defined for the base calculus. The set of kind schemes appropriate for our calculus is:

$K ::=$ U $\mid$ P $\mid$ <$l{:}T,\ldots,l{:}T$>

where U is the universal kind scheme, P is the partial kind scheme and $<l:\sigma,\ldots,l:\sigma>$ stands for record kind schemes. The kinding relation is given as follows:

$$
\begin{aligned}
T &:: \mathtt{U} \\
\mathcal{P}(\kappa) &:: \mathtt{P} \\
t^{<l_1:T_1,\ldots,l_n:T_n,\ldots>} &:: <l_1:T_1,\ldots,l_n:T_n> \\
u^{<l_1:T_1,\ldots,l_n:T_n,\ldots>} &:: <l_1:T_1,\ldots,l_n:T_n> \\
[l_1:T_1,\ldots,l_n:T_n,\ldots] &:: <l_1:T_1,\ldots,l_n:T_n> \\
\mathcal{P}(<[l_1:\tau_1,\ \ldots,l_n:\tau_n,\ldots]>) &:: <l_1:\tau_1,\ldots,l_n:\tau_n> \\
\mathcal{P}(<l_1:\tau_1,\ \ldots,l_n:\tau_n,\ldots>) &:: <l_1:\tau_1,\ldots,l_n:\tau_n> \\
\mathcal{P}(<l_1:\sigma_1,\ \ldots,l_n:\sigma_n,\ldots>^{eq}) &:: <l_1:\sigma_1,\ldots,l_n:\sigma_n>
\end{aligned}
$$

The notion of substitutions is refined to represent kind constraints. A *kind preserving substitution* θ is a function from the set of kinded type variables to kinded type schemes such that $\theta(t^K) \neq t^K$ for only finitely many t^K, it maps eqtype variables to eqtype schemes, and $\theta(t^K) :: \theta(K)$ for all t^K. The notion of unifier and most general unifiers are defined as usual.

Robinson's unification algorithm is refined to kinded type schemes:

Proposition 4 *There is an algorithm $\mathcal{U}$ which computes a most general kind preserving unifier of a given set of pairs (equations) of kinded type schemes if one exists; otherwise it reports failure.* ∎

This is a simple extension of a result shown in [Oho90]. ∎

In order to represent the constraints associated with the rules (union) and (fuse), we have to introduce explicit conditions on substitutions of type variables. The following definitions are straightforward adaptation of the method developed in [OB88]. Define *conditions* as formula of the forms: $T = lub(T,T)$ or $T = glb(T,T)$. We say that a kind preserving substitution θ *is ground for* X if, for any type variable t^K in X, $\theta(t^K)$ is a type. A substitution θ ground for c *satisfies* a condition c, denoted by $\theta \models c$, if

1. $c \equiv T_1 = lub(T_2,T_3)$ and $\theta(T_1) = \theta(T_2) \sqcup \theta(T_3)$, or
2. $c \equiv T_1 = glb(T_2,T_3)$ and $\theta(T_1) = \theta(T_2) \sqcap \theta(T_3)$.

Let C be a set of conditions, Σ be an assignment of kinded type schemes to variables. Define $\theta \models C$ iff $\theta \models c$ for all $c \in C$. A *conditional principal typing scheme* is a formula of the form $C, \Sigma \rhd e : T$ such that $\mathcal{A} \rhd e : \tau$ iff there is a kind preserving substitution θ such that $\theta \models C$, $\mathcal{A}(x) = \theta(\Sigma(x))$ for all $x \in dom(\Sigma)$ and $\tau = \theta(T)$. Since the definition of conditional typing schemes and the conditions have similar properties of those in [OB88], the following theorem can be proved similarly.

Theorem 2 *There is an algorithm $\mathcal{P}$ which, given an untyped term e, returns either (C,Σ,T) of failure such that if $\mathcal{P}(e) = (C,\Sigma,T)$ then $C, \Sigma \rhd e : T$ is a principal conditional typing scheme otherwise e has no typing.* ∎

As in [OB88], however, this theorem does not completely solve the type-checking problem because of possible unsatisfiability of the set of conditions. Unfortunately, the satisfiability checking turns out to be a hard problem. Since the condition of the form $T = lub(T,T)$ has the identical structure to those we have used for the database operation **join** in [OB88], the complexity of the complete type-checking problem is as hard

as in [OB88], which is NP-complete. A practical solution is to *delay* the satisfiability-checking until all the type variables in conditions are instantiated with ground types. Once all type variables are instantiated with ground types, the satisfiability can be efficiently checked. To achieve this, we require the type specification for {} if it is used in the argument of **union** with other dynamic values. Violation of this requirement can be easily detected by the compiler. Under this restriction, it is easily shown that **union** and **fuse** are evaluated only if the arguments have ground types and therefore this strategy prevents all run time type errors. Since we have the singleton constructor and **union**, the most common situation when {} is needed as an argument of **merge** is inside of **reduce**. Taking this into account, the typing rule for **reduce**(f,op,z,S) is defined so that the type of op must be the product of the same type.

We now show examples to demonstrate how we achieve inheritance through ML style polymorphism. We omit the kind tag `U`. Suppose we have two sets:

```
Employees : {P(<Name:string,Address:string,Salary:int>)}
Students  : {P(<Name:string,Address:string,Advisor:string>)}
```

and define the functions

```
fun advisors S = reduce(λx. x.Advisor),union,{},S);
fun add_saraly S = reduce(λx. modify(x,Salary,x.Salary + 500),union,{},S);
```

Since the type system infers the following schemes:

$$\emptyset,\emptyset \ \triangleright \ \texttt{advisor} : \{t_1^{<Advisor:t_2>}\} \rightarrow \{t_2\}$$
$$\emptyset,\emptyset \ \triangleright \ \texttt{add\_saraly} : \{t_1^{<Salary:int>}\} \rightarrow \{t_1^{<Salary:int>}\}$$

they can be applicable to **Students** and **Employees** respectively by kinded preserving instantiation. We can think of these polymorphic functions as "methods" applicable to **Students** and **Employee** respectively. As we have advocated, we expect both of the two functions to be applicable to the intersection of the two sets. In the previous section, we have defined the function **intersect**. For the untyped code of this function, the type system infers the following scheme:

$$\{t_1 = \texttt{lub}(t_2,t_3)\},\emptyset \ \triangleright \ \texttt{intersect} : \{t_2\}*\{t_3\} \rightarrow \{t_1\}$$

When applied to the two set above, this function yields the following typing:

```
SupportedStudents = intersect(Students,Employees) :
                    {P(<Name:string,Address:string,Salary:int,Advisor:string>)}
```

Our polymorphism indeed allows us to apply both of the **add_salary** and **advisor** to the above set without any loss of type information. For example, the type of **add_salary(SupportedStudents)** is the same as that of **SupportedStudents** as we desired.

4 Conclusions and Further Investigation

The combination of partial types with the appropriate "bulk" data type provides a method of dealing with heterogeneous collections in a statically typed language. However these two additions to the type system are almost independent of one another. The only operations in which they directly interact are **as**, **coerce** and **fuse** which return singleton or empty collections. This is a convenience that avoids the need to deal

with run-time exceptions. It would be possible to describe partial types without making use of a collection type. We have discussed two examples of collection types, lists and sets, and it appears possible to use the same techniques in connection with other bulk types, binary trees or bags, for example. What we do not yet understand is what is the general characterization of collection types? Why do partial types appear to fit naturally with certain types and not with others? This seems to call for a rather general insight into programming structures.

The ordering on types we have exploited is trivial except on partial record types. Does it need to be extended to other types? For database and object-oriented programming, the notion of "object identity", which appears to be similar to reference, is often invoked. It seems to be straightforward to extend the ordering to work on reference types. However, it is not clear that we need to extend it, say, to function types. An issue here is whether there is some useful "information" ordering on functions. There are useful extensionally defined functions in databases, hash tables and B-trees, for example. But is there a natural *partial* description of such structures as there is for records?

Finally, implementation is a serious issue. On face value this system calls for dynamic resolution of record fields. It may be that there are efficient architectures for doing this; it may also be possible to find some optimization techniques. For example, in many situations our heterogeneous collections may be interally represented by a small number of uniformly represented homogeneous collections (this would be the case in using data derived from a relational database). Operations on heterogeneous collections could then be decomposed into operations on homogeneous collections, for which relational database technology has provided us with optimization techniques.

References

[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989.

[BJO90] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, To Appear. Available as a technical report from Department of Computer and Information Science, University of Pennsylvania, 1989.

[BTBO89] V. Breazu-Tannen, P. Buneman, and A. Ohori. Can object-oriented databases be statically typed? In *Proc. 2nd International Workshop on Database Programming Languages*, pages 226 – 237, Gleneden Beach, Oregon, June 1989. Morgan Kaufmann Publishers.

[Car86] L. Cardelli. Amber. In *Combinators and Functional Programming, Lecture Notes in Computer Science 242*, pages 21–47. Springer-Verlag, 1986.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. (Special issue devoted to Symp. on Semantics of Data Types, Sophia-Antipolis, France, 1984).

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surverys*, 17(4):471–522, December 1985.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. American Mathematical Society*, 146:29–60, December 1969.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[JM88] L. A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit90] J. C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, 1990.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988.

[Oho89] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science, Special issue dedicated to* 2^{nd} *International Conference on Database Theory (To Appear).* Available as a technical report form University of Pennsylvania, 1989.

[Oho90] A. Ohori. Extending polymorphism to records and variants. Unpublished manuscript, Preliminary abstract presented at 6^{th} Workshop on Mathematical Foundation of Programming Semantics, 1990.

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In David MacQueen, editor, *ACM Conference on Principles of Programming Languages*, 1989.

[Str87] B. Stroustrup. *The C++ programming language.* Addison-Wesley, 1987.

[Tha88] S. R. Thatte. Type inference with partial types. In 15^{th} *International Colloquium of Automata, Languages and Programming, Lecture Notes in Computer Science 317*, pages 615–629, 1988.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987.

[Wan89] M. Wand. Type inference for records concatenation and simple objects. In *Proceedings of 4th IEEE Symposim on Logic in Computer Science*, pages 92–97, 1989.