Combining Inheritance and Parametric Polymorphism in a Functional Database Language

Simon Courtenage and Alexandra Poulovassilis

Dept. of Computer Science, King's College London
Strand, London WC2R 2LS
{alex,simonc}@dcs.kcl.ac.uk

Abstract. We consider extending a functional database language to support subtyping, inheritance and method overloading. We do so by extending previous work on type inference with subtypes for the pure λ calculus to cater for structured types, ML-style parametric polymorphism and overloaded function definitions. We attach semantics to overloaded functions by developing a generalisation of best-fit pattern-matching. Although developed for a specific language, our approach is applicable to other functional database languages, for example languages with a functional data model.

1 Introduction

Two major trends in recent database research have been deductive databases and object-oriented databases. Deductive databases extend the relational data model with rules that enable the derivation of intentional relations from the stored, extensional, relations. Object-oriented databases typically start off with a semantic data model [HK87] supporting object identity and complex objects, and extend it with features such as inheritance, methods and encapsulation from object-oriented programming. Deductive and object-oriented databases are thus largely complementary and recent research has aimed at integrating them [DOO].

For reasons discussed at length in [PS91], we are interested in *functional programming* as opposed to logic programming as the basis for deductive databases. A database language, PFL [PS91] [SP91], has been developed which exhibits all the desirable features of a functional language (such as higher-order functions, static type checking and lazy evaluation) but which also supports functions that simulate the extensional and intentional relations of logic languages such as Datalog.

In common with many functional programming languages, PFL supports ML-style parametric polymorphism of types and functions. We now show how to incorporate subtyping, inheritance and overloading into PFL, thereby taking a functional programming route to the integration of deductive and object-oriented databases. Two fundamental requirements for this work have been (a) to preserve upwards-compatibility with the current version of PFL, and (b) to avoid "impedance mismatch" problems by maintaining a single semantic and operational framework. Our approach is easily applied to other functional database languages, for example languages with a functional data model such as FDL [Pou92].

We begin the paper with a brief review of the current PFL language in Section 2. We then introduce its new features in Section 3. In particular, we discuss how the type system is enriched by allowing the declaration of subtype relationships between atomic (i.e. non-structured) types, and how subtyping extends to structured types and function types. We describe how function definitions can be overloaded for different subtypes of a type, and we attach semantics to overloaded functions by developing a generalisation of the best-fit pattern-matching of [FH88]. We also briefly consider schema evolution. Type-checking remains static in the extended language but some type inference is also carried out at run time in order to perform run-time binding to overloaded functions. We give the type inference algorithm in Section 4. It is based upon Mitchell's $\mathcal{GA}$ algorithm [Mit91], with additions to infer types for constructors, "let" and "letrec" expressions and overloaded functions. In Section 5 we consider related work in the area of type inference with subtypes. We end the paper in Section 6 with a summary of our contributions and directions of further work

2 Overview of PFL

In this section we give an overview of the current PFL language, covering only aspects of the language that are prerequisites for our discussion of its new features in subsequent sections. More comprehensive accounts of PFL can be found in [PS91], [SP91].

2.1 Types

The syntax of PFL types is as follows, where σ ranges over type expressions, α over type variables and κ^n over n-ary type constructors:

$$\sigma ::= \alpha \mid \sigma_1 \rightarrow \sigma_2 \mid \kappa^n \ \sigma_1 \ \ldots \ \sigma_n$$

Types of the form $\sigma_1 \rightarrow \sigma_2$ are *function types*. Types of the form κ^0 are *atomic types* (so the set of 0-ary type constructors and the set of atomic types are identical). Types of the form $\kappa^n \ \sigma_1 \ \ldots \ \sigma_n$, where $n > 0$, are *structured types*. Below, we also use the symbols ι, κ to range over atomic types and τ, υ to range over type expressions.

The atomic types Num, Chr, $Bool$ and Str are built-in and declared as follows in PFL's standard environment:

$$\begin{array}{ll}
_type\ Num & :: Type \\
_type\ Chr & :: Type \\
_type\ Bool & :: Type \\
_type\ Str & :: Type
\end{array}$$

Also built-in are a number of type constructors for structured types:

$$
\begin{array}{ll}
\_type\ List & :: Type \rightarrow Type \\
\_type\ Set & :: Type \rightarrow Type \\
\_type\ Product2 & :: Type \rightarrow Type \rightarrow Type \\
\_type\ Product3 & :: Type \rightarrow Type \rightarrow Type \rightarrow Type \\
\ldots &
\end{array}
$$

These declarations can be read as stating that a type constructor takes one or more types as arguments and constructs a new type. PFL supports some syntactic sugar for structured types: $(\tau_1, \ldots, \tau_n)$ is synonymous with $Product_n\ \tau_1\ \ldots\ \tau_n$ for any types $\tau_1, \ldots, \tau_n$; $[\tau]$ is synonymous with $List\ \tau$ for any type τ; and $\{\tau\}$ is synonymous with $Set\ \tau$. Also, $\rightarrow$ is right-associative, so that $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ and $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ are synonymous.

New constructors for atomic and structured types can be declared at any time during the lifetime of a PFL database e.g.

$$
\begin{array}{ll}
\_type\ Person & :: Type \\
\_type\ Student & :: Type \\
\_type\ Employee & :: Type \\
\_type\ RA & :: Type \\
\_type\ Tree & :: Type \rightarrow Type
\end{array}
$$

We use the notation $\mathrm{E} :: \tau$ to indicate that an expression E has type τ. We postpone a precise definition of the syntax of expressions to Section 2.4.

2.2 Values

Both structured and atomic types are incrementally populated by declaring *value constructors* which return values of that type when applied to appropriate arguments. The 0-ary value constructors populating the built-in Num, Chr, Str and $Bool$ types are predeclared and non-deletable - these are, respectively, integer and real numbers, characters, strings, and $True$ and $False$. Also predeclared are constructors for empty lists, lists consisting of a head and tail, and tuples:

$$
\begin{array}{ll}
\_value\ [\,] & :: [a] \\
\_value\ (:) & :: a \rightarrow [a] \rightarrow [a] \\
\_value\ Tuple2 & :: a \rightarrow b \rightarrow (a, b) \\
\_value\ Tuple3 & :: a \rightarrow b \rightarrow c \rightarrow (a, b, c) \\
\ldots &
\end{array}
$$

In the above declarations, a, b, and c are type variables that can be instantiated by any type. Generally, in PFL identifiers starting with a lower-case letter are variables while identifiers starting with an upper-case letter are type or value constructors.

Some syntactic sugar is supported for structured values in that, for any expressions $\mathrm{E}_1, \ldots, \mathrm{E}_n$, an n-tuple $(\mathrm{E}_1, \ldots, \mathrm{E}_n)$ is synonymous with $Tuple_n\ \mathrm{E}_1\ \ldots$

E_n and an enumerated list $[\mathrm{E}_1, \ldots, \mathrm{E}_n]$ is synonymous with $\mathrm{E}_1 : (\ldots : (\mathrm{E}_n : [\,])\ldots)$.

New value constructors can be declared at any time during the lifetime of a PFL database e.g.

$$\begin{array}{llll}
_value\ E1 & :: Employee & _value\ E2 & :: Employee \\
_value\ S1 & :: Student & _value\ R1 & :: RA \\
_value\ Leaf & :: a \rightarrow (Tree\ a) & _value\ Node & :: (Tree\ a) \rightarrow (Tree\ a) \rightarrow (Tree\ a)
\end{array}$$

0-ary constructors such as $E1$, $E2$, $S1$, $R1$ above serve as *object identifiers* in PFL. User-declared atomic types are analogous to the non-lexical types of languages such as FDL and 0-ary constructors are analogous to non-lexical entities.

2.3 Functions

The usual arithmetic, comparison and boolean operators are built-in. Also built-in are several operators over sets, including set union, $\cup :: \{a\} \rightarrow \{a\} \rightarrow \{a\}$, and $setmap :: (a \rightarrow \{b\}) \rightarrow \{a\} \rightarrow \{b\}$. The latter distributes a function of type $a \rightarrow \{b\}$ over a set of type $\{a\}$ and returns the union of the results. Binary operators may be written either infix or prefix, and in the latter case they must be enclosed within brackets.

The user can define new functions by means of one or more equations. Equations use pattern-matching to deconstruct their arguments, where a *pattern* is either a variable or an application of an n-ary constructor to n patterns. For example, the following PFL commands specify two equations for the $foldr$ function and one equation for the sum function (note that in PFL the symbol $==$ is used for equational definitions while $=$ is the equality operator):

$$\begin{array}{ll}
_define\ foldr\ op\ e\ [\,] & == e \\
_define\ foldr\ op\ e\ (x : xs) & == op\ x\ (foldr\ op\ e\ xs) \\
_define\ sum\ xs & == foldr\ (+)\ 0\ xs
\end{array}$$

Single-equation definitions are simply syntactic sugar for named λ- abstractions e.g. the definition for sum above states that $sum = \lambda xs.foldr\ (+)\ 0\ xs$.

Multi-equation definitions also translate into a single expression - the reader is referred to [PJ87], [FH88] for a detailed description of the process. For example, the single expression that defines $foldr$ above is

$$\begin{array}{ll}
foldr\ = \lambda op.\lambda e.\lambda l. & (\lambda op.\lambda e.\lambda[\,].e)\ op\ e\ l\ || \\
& (\lambda op.\lambda e.\lambda(x : xs).op\ x\ (foldr\ op\ e\ xs))\ op\ e\ l\ || \\
& error
\end{array}$$

where:

- $|| :: a \rightarrow a \rightarrow a$ is a built-in binary infix operator (pronounced “fatbar”) which returns its first argument if this is not the value $Fail$ and its second argument otherwise;
- $error :: a$ is a built-in function which generates a run-time error when invoked; and

– $Fail :: a$ is a built-in value of unconstrained type which is used to denote a pattern-matching failure.

Thus, when at run time $foldr$ is applied to three arguments op, e and l, if l matches the pattern $[\,]$ (i.e. it is the empty list) then the first definition is used, otherwise if l matches the pattern $(x : xs)$ the second definition is used, otherwise the third definition is used.

λ-abstractions with patterns as their bound argument, such as $\lambda[\,].e$ and $\lambda(x : xs).op\ x\ (foldr\ op\ e\ xs)$ above, are just syntactic sugar for ordinary λ-abstractions of the form $\lambda x.$E. In particular, corresponding to each n-ary constructor $c :: \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ is a built-in function $unpack\_c :: (\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow a) \rightarrow \tau \rightarrow a$ and λ-abstractions of the form $\lambda(c\ p_1\ \ldots\ p_n).$E translate into $unpack\_c\ (\lambda p_1 \ldots . \lambda p_n.\mathrm{E})$. The semantics of $unpack\_c$ are

$$\begin{aligned} unpack\_c\ f\ (c\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n) &= f\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n \\ unpack\_c\ f\ (c'\ \mathrm{E}_1\ \ldots\ \mathrm{E}_m) &= Fail \qquad\qquad if\ c' \neq c \\ unpack\_c\ f\ \bot &= \bot \end{aligned}$$

where $\bot$ denotes a non-terminating computation. In other words, when at run time $unpack\_c$ is applied to two arguments, f and E, E is evaluated as far as is necessary to determine its constructor; if this evaluation fails to terminate, the result is a non-terminating computation; if E evaluates to an expression of the form $c\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n$ the result is $f\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n$; otherwise the result is $Fail$.

The type of any equationally-defined function is inferred automatically by the system and the user is warned if the function is not well-typed.

Bulk data is stored within extensionally defined sets termed *data functions* whose types are explicitly declared by the user [1] e.g.

$$\begin{aligned} &\_datafun\ courses :: \{(Student, Course)\} \\ &\_datafun\ salary\ \ :: \{(Employee, Num)\} \\ &\_datafun\ income\ :: \{(Person, Source, Num)\} \end{aligned}$$

For example, the following functions return respectively the courses of a student, the salary of an employee, the set of $(source, income)$ tuples for a person, and the total income for a person [2]:

$$\begin{aligned} &\_define\ attends\ s\ \ == \{c \mid (q, c) \leftarrow courses;\ q = s\} \\ &\_define\ earns\ e\ \ \ \ == one\ \{s \mid (q, s) \leftarrow salary;\ q = e\} \\ &\_define\ incomes\ p == \{(s, i) \mid (q, s, i) \leftarrow income;\ q = p\} \\ &\_define\ totalInc\ p == sum\ [i \mid (s, i) \leftarrow set\_to\_list\ (incomes\ p)] \end{aligned}$$

[1] This is actually a simplification of how the current implementation of PFL handles bulk data; in particular, data functions cannot be used directly but only via functions termed *selectors* [SP91]. However, in the interests of simplicity, we assume in this paper that data functions are directly usable: the type system with which we are concerned copes equally well with selector functions and data functions.

[2] See [PS94] for a discussion of the function *set_to_list* which deterministically converts a set to a list. The function *one* in *earns* is equal to $head \circ set\_to\_list$

We note the use of set and list comprehensions in the above definitions. As described at length elsewhere [PJ87] [PS94], these constructs are merely syntactic sugar for applications of *setmap* and *flatmap* (the analogue of *setmap* for lists).

The initial value of any data function is the empty set. This value is updateable by means of two built-in functions which respectively add a value to a data function and delete a value from a data function, returning the updated data function [3]:

$inc :: a \rightarrow \{a\} \rightarrow \{a\}$
$exc :: a \rightarrow \{a\} \rightarrow \{a\}$

2.4 Type-Checking and Evaluation

PFL expressions are translated into the following kernel syntax for the purposes of type-checking and evaluation, where $x \in Variables$, including the names of equationally-defined functions, and $c \in Constants$, including the names of constructors, built-in functions and data functions:

$$
\begin{aligned}
\mathrm{E} ::= {} & x \ \mid \ c \ \mid \ \mathrm{E}_1\mathrm{E}_2 \ \mid \ \lambda x.\mathrm{E} \ \mid \\
& let \ \ x \ = \ \mathrm{E}_1 \ in \ \mathrm{E} \ \mid \\
& letrec \ \ (x_1, \ldots, x_n) \ = \ (\mathrm{E}_1, \ldots, \mathrm{E}_n) \ \ in \ \ \mathrm{E}
\end{aligned}
$$

In order to type-check and evaluate a query, E, the *call graph* of E, $CG(\mathrm{E})$, is first constructed i.e. the set of equationally-defined functions that are either directly or indirectly referenced by E. From the discussion in the previous section, $CG(\mathrm{E})$ is a set of definitions $\{f_1 = \mathrm{E}_1, \ldots, f_r = \mathrm{E}_r\}$. This set is next partitioned into strata, $S_1, \ldots, S_m$, such that every function defined in a stratum S_i either directly or indirectly references every other function defined in S_i, and does not reference any function defined in a stratum S_j with $j > i$. Each stratum S_i is finally translated into a "let" or "letrec" clause, L_i, as follows. If S_i consists of just one non-recursively defined function, $S_i = \{f_{i,1} = \mathrm{E}_{i,1}\}$, then L_i is

$$let \ \ f_{i,1} = \mathrm{E}_{i,1}$$

otherwise if S_i consists of one or more mutually recursively defined functions, $S_i = \{f_{i,1} = \mathrm{E}_{i,1}, \ldots, f_{i,r_i} = \mathrm{E}_{i,r_i}\}$ then L_i is

$$letrec \ \ (f_{i,1}, \ldots, f_{i,r_i}) = (\mathrm{E}_{i,1}, \ldots, \mathrm{E}_{i,r_i})$$

[3] This is again a simplification of how updates are handled in PFL since, as they stand, *inc* and *exc* have the drawback that loss of referential transparency can occur, and that the evaluation order of expressions must be known to the programmer in order to sequence evaluations of *inc* and *exc*. [SS95] discusses how to overcome these problems. The type system with which we concerned in this paper is largely orthogonal to the way in which updates are accommodated.

The overall expression that will be type-checked and evaluated given a query E is thus

$$L_1 \;\; in \;\; L_2 \;\; in \;\; \ldots \;\; L_m \;\; in \;\; \mathrm{E}$$

Type inference is performed essentially by means of Milner's W algorithm and query evaluation is performed by means of the standard technique of graph reduction used in many functional programming languages. We refer the reader to [PJ87] and [FH88] for a detailed description of both issues.

3 New Features

We are now ready to introduce the new features of PFL. We begin with a description of how *subtyping* is supported by the, possibly incremental, declaration of an inheritance graph (Section 3.1). We then discuss the implications of subtyping on constructors, built-in functions and equationally-defined functions (Sections 3.2, 3.3 and 3.4). Overloading of function definitions is now supported by allowing variables annotated with atomic types to appear in equation left-hand sides. Overloaded functions are analogous to methods in object-oriented languages and we define their semantics in Section 3.5. In Section 3.6 we describe how the extended language is translated into a kernel syntax for type-checking and evaluation. Finally, in Section 3.7 we discuss the issue of schema evolution.

3.1 Subtypes

The first new feature is that atomic types can participate in an *inheritance graph*. For example, given the atomic types declared in Section 2.1, the usual inheritance graph of $RA < Employee, RA < Student, Employee < Person, Student < Person$ is defined by the following commands:

$\_type\ RA < Employee$
$\_type\ RA < Student$
$\_type\ Employee < Person$
$\_type\ Student < Person$

Arcs can be added to the inheritance graph at any time during the lifetime of a PFL database. Before proceeding with the insertion of a new arc, $\iota < \kappa$, into the current inheritance graph, G, we:

(i) verify that the subtype relationship $\kappa < \iota$ is not inferable from G, thereby ensuring that no cycles are introduced into G;
(ii) verify that for each arc $\iota' < \kappa' \;\in\; G \cup \{\iota < \kappa\}$ the subtype relationship $\iota' < \kappa'$ is not inferable from $(G \cup \{\iota < \kappa\}) - \{\iota' < \kappa'\}$, thereby ensuring that every arc in the new inheritance graph is non-redundant.

The insertion of a new arc $\iota < \kappa$ into the inheritance graph has two effects: it causes the extent of κ to expand to include the extent of ι, and it induces further subtype relationships between types. The inference rules for deducing new subtype relationships from a given set of subtype relationships C are as follows:

$$\frac{}{\mathrm{C} \vdash \sigma < \sigma}\ \text{Reflexivity}$$

$$\frac{\mathrm{C} \vdash \sigma < \tau \qquad \mathrm{C} \vdash \tau < \upsilon}{\mathrm{C} \vdash \sigma < \upsilon}\ \text{Transitivity}$$

$$\frac{\mathrm{C} \vdash \tau_1 < \sigma_1 \qquad \mathrm{C} \vdash \sigma_2 < \tau_2}{\mathrm{C} \vdash \sigma_1 \rightarrow \sigma_2 < \tau_1 \rightarrow \tau_2}\ \text{Function types}$$

$$\frac{\mathrm{C} \vdash \sigma_1 < \tau_1, \ldots, \sigma_n < \tau_n}{\mathrm{C} \vdash \kappa^n \sigma_1 \ldots \sigma_n < \kappa^n \tau_1 \ldots \tau_n}\ \text{Structured types}$$

We notice from these rules that the $\rightarrow$ type constructor is anti-monotonic in its first argument while all other type constructors are monotonic in all arguments. These monotonicity properties ensure the semantic soundness of subtypes - see [CW85] for a discussion.

3.2 Values

In the presence of subtypes, constructors now have richer types than before. We term the declared types of constructors *Curry types* and the enriched types *Mitchell types*, after [Mit91]. In particular, a Mitchell type is a pair (C,τ) where C is a *coercion set* consisting of subtype assertions between atomic types and type variables and where τ is a type expression that references no atomic types. For example, the Mitchell type of the constructor $E1$ of Section 2.2 is $(\{Employee < a\}, a)$, meaning that $E1$ has type $Employee$ and also any supertype thereof.

The Curry types declared for constructors are straightforwardly converted into the corresponding Mitchell types. In particular, the Mitchell type corresponding to a Curry type τ is given by EXPAND($\tau, True$) where the algorithm EXPAND is defined below over the syntax of type expressions. The second argument to EXPAND indicates whether the current sub-expression appears covariantly in the overall type expression. The definition of EXPAND for a function type negates this value for the argument type:

$$\begin{array}{ll}
\text{EXPAND}(\kappa, True) & = (\{\kappa < \beta\}, \beta) \\
\text{EXPAND}(\kappa, False) & = (\{\beta < \kappa\}, \beta) \\
\text{EXPAND}(\alpha, True) & = (\{\alpha < \beta\}, \beta) \\
\text{EXPAND}(\alpha, False) & = (\{\beta < \alpha\}, \beta)
\end{array}$$

where β is a new type variable

$$
\begin{aligned}
&\text{EXPAND}(\sigma_1 \to \sigma_2, covariant) = (\text{C}_1 \cup \text{C}_2, \tau_1 \to \tau_2)\\
&\quad \text{where}\\
&\quad (\text{C}_1, \tau_1) = \text{EXPAND}(\sigma_1, not(covariant)),\\
&\quad (\text{C}_2, \tau_2) = \text{EXPAND}(\sigma_2, covariant)\\
&\text{EXPAND}(\kappa^n\ \sigma_1 \ldots \sigma_n, covariant) = (\text{C}_1 \cup \ldots \cup \text{C}_n, \kappa^n\ \tau_1 \ldots \tau_n)\\
&\quad \text{where}\\
&\quad (\text{C}_1, \tau_1) = \text{EXPAND}(\sigma_1, covariant)\\
&\quad \ldots\\
&\quad (\text{C}_n, \tau_n) = \text{EXPAND}(\sigma_n, covariant)
\end{aligned}
$$

For example, the type declared for the constructor (:) in Section 2.2 was $a \to [a] \to [a]$ and

$$\text{EXPAND}(a \to [a] \to [a], True) = (\{b < a, c < a, a < d\}, b \to [c] \to [d]).$$

Thus whereas before (:) constructed lists of elements all of the same type a, it now constructs lists of elements all with a common supertype a, and the resultant list possesses all types $[d]$ such that $a < d$. For example, $[E1, S1]$ is now a list of type $[Person]$ whereas before it would have been incorrectly typed.

3.3 Built-in functions

In the presence of subtypes, built-in functions generally have richer types than before. Unlike our treatment of constructors above, the Mitchell type of a built-in function cannot in general be determined from its Curry type but instead has to be declared explicitly by the implementor. The following are some of the more interesting changes to the built-in functions:

The *asaκ* family of functions. For every atomic type κ, we now support a new built-in function $asa\kappa :: (\{a < \kappa\}, a \to \kappa)$ which generalises the type of an expression from a subtype of κ to κ itself. As we will see below, this family of functions is useful for generalising the type of an expression in order to bind at run time to the method definition for a supertype.

The arithmetic operators. We can now refine PFL's built-in Num type with a subtype structure representing different kinds of numbers and we can enrich the built-in arithmetic operators appropriately. For example, if $Int < Real$ and $Real < Num$ we can implement an addition operator $(+) :: (\{a < Num, b < Num, a < c, b < c\}, a \to b \to c)$ in which case $1 + 2 :: (\{Int < c, c)$ while $1.5 + 2 :: (\{Real < c\}, c)$.

The *unpack_c* family of functions. Analogously to constructors, the *unpack_c* functions of Section 2.3 now have types obtained by applying EXPAND to their Curry type. For example, the Curry types of *unpack_*[ ], *unpack_*(:) and $unpack\_Tuple2$ were

$$\begin{array}{ll} unpack\_[\,] & :: a \rightarrow [b] \rightarrow a \\ unpack\_(:) & :: (a \rightarrow [a] \rightarrow b) \rightarrow [a] \rightarrow b \\ unpack\_Tuple2 & :: (a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c \end{array}$$

while their Mitchell types (after some simplification of the coercion sets) are

$$\begin{array}{ll} unpack\_[\,] & :: (\{a < c\}, a \rightarrow [b] \rightarrow c) \\ unpack\_(:) & :: (\{f < c, f < d, e < g\}, (c \rightarrow [d] \rightarrow e) \rightarrow [f] \rightarrow g) \\ unpack\_Tuple2 & :: (\{g < d, h < e, f < i\}, (d \rightarrow e \rightarrow f) \rightarrow (g, h) \rightarrow i) \end{array}$$

For example, the expression $\lambda(e, s).(earns\ e, attends\ s)$ translates into
$unpack\_Tuple2\ (\lambda e.\lambda s.(earns\ e, attends\ s))$
and has type $(\{a < Employee, b < Student, Num < c, Course < d\}, (a, b) \rightarrow (c, \{d\}))$, while the expression $\lambda[e, s].(earns\ e, attends\ s)$ translates into
$unpack\_(:)\ (\lambda e.unpack\_(:)\ (\lambda s.unpack\_[\,]\ (earns\ e, attends\ s)))$
and has type $(\{a < RA, Num < b, Course < c\}, [a] \rightarrow (b, \{c\}))$. Notice that this latter expression would not be typable without subtyping.

The *inc* and *exc* functions. These now have type $(\{a < b, b < c\}, a \rightarrow \{b\} \rightarrow \{c\})$, which means that subtype values can now appear within data functions. For example, given the data functions declared in Section 2.3, the following inclusions are now valid:

$$\begin{array}{l} inc\ (E1, PropertyRent, 2000)\ \ income \\ inc\ (S1, Demonstrating, 50)\ \ income \\ inc\ (R1, DepositAccount, 200)\ \ income \\ inc\ (R1, 15000)\ \ salary \end{array}$$

In contrast, without subtyping, if we had wanted to record incomes for employees, students and RAs we could not have used the data function *income* as above but would have required three further data functions: $empIncome :: \{(Employee, Source, Num)\}$, $stuIncome :: \{(Student, Source, Num)\}$ and $raIncome :: \{(RA, Source, Num)\}$. Similarly, we could not have used the data function *salary* for RAs but would have required a further data function $raSalary :: \{(RA, Num)\}$.

3.4 Equationally-defined functions

Just as built-in functions, equationally-defined functions too can have richer types in the presence of subtypes. For example, the functions *incomes* and *totalInc* defined in Section 2.3 can now be used to determine the income of employees, students and RAs, as well as that of persons.

We now also extend the syntax of equation left hand sides to allow variables to be optionally annotated with atomic types. This extension allows functions to be *overloaded* for different subtypes of a type. For example, suppose we wish to define a function which computes the taxable income of a person in terms of their salary, if any, and other sources of income. Without subtyping, we would have had to define four different functions to do this, using the various data functions for persons, employees, students and RAs. With subtyping and overloading we need define only one function:

$$\begin{array}{l} \_define\ taxInc\ p :: Person \quad == totalInc\ p \\ \_define\ taxInc\ p :: Employee == (taxInc\ (asaPerson\ p)) + (earns\ p) \end{array}$$

The type inferred for $taxInc$ is $(\{a < Person, Num < b\}, a \rightarrow b)$. If at run time $taxInc$ is applied to an argument of type $(\{Employee < a\}, a)$ or $(\{RA < a\}, a)$ the second equation will be selected as a *more specific* match (see below) for the given argument than the first equation. Otherwise if $taxInc$ is applied to an argument of type $(\{Student < a\}, a)$ or $(\{Person < a\}, a)$ the first equation will be chosen as the only match. Notice in the second equation the use of $asaPerson$ to generalise the type of p to $Person$ and thus to pick up the first equation of $taxInc$ for p rather than the second equation.

Annotated variables can appear anywhere within patterns. In particular, functions over structured types and multiple arguments can also be overloaded. For example, $countStudents$ counts the number of students within a list of persons:

$$\begin{array}{lll} \_define\ countStudents\ [\,] & == & 0 \\ \_define\ countStudents\ (x :: Student : xs) & == & 1\ +\ (countStudents\ xs) \\ \_define\ countStudents\ (x :: Person : xs) & == & countStudents\ xs \end{array}$$

and *same* checks whether two person "objects" have the same "value" over their common attributes:

$$\begin{array}{l} \_define\ same\ x :: Person\ y :: Person \quad == incomes\ x\ =\ incomes\ y \\ \_define\ same\ x :: Student\ y :: Student \quad == \\ \qquad (same\ (asaPerson\ x)\ (asaPerson\ y))\ \&\ (attends\ x\ =\ attends\ y) \\ \_define\ same\ x :: Employee\ y :: Employee == \\ \qquad (same\ (asaPerson\ x)\ (asaPerson\ y))\ \&\ (earns\ x\ =\ earns\ y) \\ \_define\ same\ x :: RA\ y :: RA == \\ \qquad (same\ (asaStudent\ x)\ (asaStudent\ y))\ \& \\ \qquad (same\ (asaEmployee\ x)\ (asaEmployee\ y)) \end{array}$$

3.5 Semantics of Equationally-defined Functions

As before, multi-equation function definitions must be translated into a single expression. To cater for the presence of annotated variables in equation left hand sides we have developed an extension of the *best-fit pattern-matching* of [FH88] which we now describe. For the purposes of this description, n-ary functions are treated as functions of a single n-tuple. Thus for any function f we have a set of equations

$$f\ p_1 == \mathrm{E}_1\ ,\ f\ p_2 == \mathrm{E}_2\ ,\ \ldots\ ,\ f\ p_r == \mathrm{E}_r$$

which are to be ordered according to some criterion. The criterion we use is that of the relative *specificity* of the patterns $p_1, \ldots, p_r$, defined as follows:

Definition 1. A pattern p_i is *more specific* than a pattern p_j, written $p_i \sqsubset p_j$, or *equally specific* to a pattern p_j, written $p_i \sim p_j$, according to the following criteria:

x	$\sim$	y	for any variables x and y
$x :: \iota$	$\sqsubset$	y	
$x :: \iota$	$\sim$	$y :: \iota$	
$x :: \iota$	$\sqsubset$	$y :: \kappa$	if $\iota \neq \kappa$ and $\iota < \kappa$
$x :: \iota$	$\sim$	$y :: \kappa$	if $\iota \not< \kappa$, $\kappa \not< \iota$ and ι and κ have a common subtype
$c\ p_1\ \ldots\ p_n$	$\sqsubset$	x	for any constructor c and variable x
c	$\sqsubset$	$x{::}\kappa$	if the declared type of c is ι and $\iota < \kappa$
$c\ p_1\ \ldots\ p_n$	$\sim$	$c\ q_1\ \ldots\ q_n$	if $p_i \sim q_i$ for all $1 \leq i \leq n$
$c\ p_1\ \ldots\ p_n$	$\sqsubset$	$c\ q_1\ \ldots\ q_n$	if $p_i \sqsubset q_i$ or $p_i \sim q_i$ for all $1 \leq i \leq n$ and $p_i \sqsubset q_i$ for at least one $1 \leq i \leq n$

However, ordering equations by specificity of their left hand sides is not in general sufficient to yield an unambiguous definition. Consider for example the following ordered definition in which neither of the first two equations has a more specific left hand side than the other and yet both equations match arguments of type (RA, RA):

$$
\begin{array}{ll}
f\ (x :: Employee, y :: Employee) & == \ldots \\
f\ (x :: Student, y :: Student) & == \ldots \\
f\ (x :: Person, y :: Person) & == \ldots
\end{array}
$$

Clearly what is needed here is a fourth equation with a left hand side of $(x :: RA, y :: RA)$ which is more specific than both $(x :: Employee, y :: Employee)$ and $(x :: Student, y :: Student)$ c.f. the function *same* of the previous section.

Such ambiguities are of course analogous to multiple inheritance conflicts in object-oriented languages. In our case we verify that all equational definitions are *unambiguous* (see Definition 2 below), notifying the user of any ambiguities and not allowing ambiguous functions to be invoked in queries.

Definition 2. Two patterns p_i and p_j *overlap* if there exists a third pattern p_k that is more specific than either. A set of patterns p_1, p_2, ... p_r is *unambiguous* if for any two overlapping patterns p_i and p_j such that neither is more specific than the other, there exists a set of patterns $p_{k,1}$, $p_{k,2}$, ..., $p_{k,s}$ such that any value which matches p_i and p_j also matches one of $p_{k,1}$, $p_{k,2}$, ..., $p_{k,s}$.

This is a generalisation of ordinary best-fit pattern-matching where a *single* pattern p_k is sufficient to disambiguate two overlapping patterns p_i and p_j. This is not possible in the presence of subtyping. For example, consider the atomic types A, B, C, D such that $C < A, C < B, D < A, D < B$ and the patterns

$x :: A, x :: B$. These patterns are overlapping since expressions of type C and D are matched by both, but the patterns can only be disambiguated by two further patterns $x :: C, x :: D$.

3.6 Type-Checking and Evaluation

As before, given a query to type-check and evaluate, its call graph is constructed and a single expression in the kernel language is derived from it. However, the kernel syntax is now extended to:

$$
\begin{aligned}
\mathrm{E} ::= {} & x \mid c \mid \mathrm{E}_1\mathrm{E}_2 \mid \lambda x.\mathrm{E} \mid \lambda x :: \kappa.\mathrm{E} \mid \\
& \mathit{let}\ \ x\ =\ \mathrm{E}_1\ \mathit{in}\ \mathrm{E} \mid \\
& \mathit{letrec}\ \ (x_1, \ldots, x_n)\ =\ (\mathrm{E}_1, \ldots, \mathrm{E}_n)\ \ \mathit{in}\ \ \mathrm{E} \\
& \mathit{method}\ \ \mathrm{E}_1,\ \ldots,\ \mathrm{E}_n
\end{aligned}
$$

The two additions to the previous kernel syntax of Section 2.4 are λ-abstractions of the form $\lambda x :: \kappa.\mathrm{E}$ and "method" clauses. The former are needed in order to translate patterns containing annotated variables while the latter are needed in order to type-check overloaded functions. In particular, suppose the equations defining an n-ary function f are

$$
\begin{aligned}
& f\ p_{1,1}\ \ldots\ p_{1,n} == \mathit{rhs}_1 \\
& f\ p_{2,1}\ \ldots\ p_{2,n} == \mathit{rhs}_2\ \ldots \\
& f\ p_{r,1}\ \ldots\ p_{r,n} == \mathit{rhs}_r
\end{aligned}
$$

ordered so that if $(p_{i,1},\ \ldots p_{i,n}) \sqsubset (p_{j,1},\ \ldots p_{j,n})$ the i^{th} equation precedes the j^{th} one. Then, the single expression defining f is

$$
f = \mathit{method}\ \lambda p_{1,1}.\ldots.\lambda p_{1,n}.\mathit{rhs}_1,\ \lambda p_{2,1}.\ldots.\lambda p_{2,n}.\mathit{rhs}_2, \ldots, \lambda p_{r,1}.\ldots.\lambda p_{r,n}.\mathit{rhs}_r
$$

Finally, the *unpack_c* functions can be used exactly as before to simplify λ-abstractions.

We discuss the new type inference algorithm in Section 4 below. After a query has been type-checked, and before it is evaluated, all expressions of the form

$$
\mathit{method}\ \ \mathrm{E}_1, \mathrm{E}_2,\ \ldots\ , \mathrm{E}_r
$$

are, as before, translated into expressions invoking the $||$ operator, assuming here the function being defined is an n-ary one and that $x_1, \ldots, x_n$ do not occur free in $\mathrm{E}_1, \ldots, \mathrm{E}_r$:

$$
\lambda x_1.\ldots.\lambda x_n.\mathrm{E}_1\ x_1\ \ldots\ x_n\ ||\ \mathrm{E}_2\ x_1\ \ldots\ x_n\ ||\ \ldots\ ||\ \mathrm{E}_r\ x_1\ \ldots\ x_n\ ||\ \mathit{error}
$$

Thus, for the purposes of evaluating the extended kernel language, it only remains to define the semantics of expressions of the form $\lambda x :: \kappa.\mathrm{E}$. Since these semantics invoke the type inference algorithm in order to perform run-time binding, we postpone this discussion to Section 4.4 below.

3.7 Schema Evolution

We next consider the issue of schema evolution and, in particular, the addition and removal of arcs in the inheritance graph and the specialisation and generalisation of the types of object identifiers (i.e. 0-ary constructors of atomic type). Due to space constraints, we can only give a brief discussion here and postpone a fuller treatment to a future paper.

The addition of a new arc into the inheritance graph requires no special type-checking. This is because it cannot invalidate the current value of any data function, and because equationally-defined functions are type-checked in any case prior to their use in a query.

The removal of an existing arc from the inheritance graph is not so simple since data functions may be rendered type-incorrect (again we do not need to worry about equationally-defined functions since these are typed-checked before they are used in any query). In particular, the removal of an arc $\iota < \kappa$ may invalidate any data function $f :: \sigma$ such that σ references κ or any supertype thereof, so we only allow the deletion of the arc if there are no such data functions. We can determine this by querying the "dependency" information which is maintained in any case by PFL for use in validating the declaration and deletion of type and value constructors. Broadly speaking, this information consists of a set of triples of the form (id_1, id_2, n), meaning that id_1 references id_2 n times, which are stored in a hash file (see [PS91] for details).

As with the insertion of a new arc into the inheritance graph, the specialisation of the type of an object identifier does not require any extra type-checking since its use within any data function remains valid and since equationally-defined functions will be type-checked as needed.

Conversely, if a constructor c currently of (Curry) type ι is generalised to be of type κ (where $\iota < \kappa$), the occurrence of c within a data function may render it type-incorrect. In particular, if the type expected for the occurrence is κ' (i.e. type of the data function is of the form $\{\ldots\kappa'\ldots\}$) then $\kappa < \kappa'$ must hold. One way to support this validation efficiently is to extend the dependency information described above to be quadruples of the form $(id_1, id_2, expected\_type, n)$. We are currently evaluating this, and other, alternatives.

4 Type Checking the Extended Language

In this section we discuss how types are inferred for expressions in the extended kernel syntax of Section 3.6. We first review some preliminary terminology in Section 4.1. We then define our type inference algorithm, $\mathcal{M}$, in Section 4.2, considering in turn each category of expressions in the kernel syntax. $\mathcal{M}$ is based on the $\mathcal{GA}$ algorithm of [Mit91] which performs type inference with atomic subtyping for the pure λ calculus i.e. for expressions of the form $\mathrm{E} ::= x \mid \mathrm{E}_1\mathrm{E}_2 \mid \lambda x.\mathrm{E}$. We thus extend GA to infer types for constants, "let", "letrec" and "method" clauses, and λ-abstractions whose bound variable is annotated with an atomic type. The coercions inferred by $\mathcal{GA}$, and hence by $\mathcal{M}$, may be inconsistent or unsatisfiable with respect to the predeclared inheritance graph. We thus apply a further algorithm COERCE_CHECK, defined in Section 4.3, to verify the consistency and satisfiability of the coercion sets inferred at each step of $\mathcal{M}$. Finally, in Section 4.4 we define the semantics of λ-abstractions whose bound variable is annotated with an atomic type.

4.1 Terminology

We will require the following terminology (mainly taken from [Mit91]):

A *coercion set* C is a set of subtype assertions between arbitrary types. An *atomic coercion set* C is a set of subtype assertions between type variables and atomic types only. A *type environment* A is a set of typing statements of the form $x :: \tau$, where x is a variable and τ a type expression.

$\mathrm{A}-\{x_1, \ldots, x_n\}$ denotes the type environment obtained from A by removing the existing typing for each variable $x_1, \ldots, x_n$, if any. $\mathrm{A}+\{x_1 :: \tau_1, \ldots, x_n :: \tau_n\}$ denotes the type environment $(\mathrm{A}-\{x_1, \ldots, x_n\}) \cup \{x_1 :: \tau_1, \ldots, x_n :: \tau_n\}$.

A *substitution* is a mapping of type variables $\alpha_1, \ldots, \alpha_n$ to type expressions $\tau_1, \ldots, \tau_n$, respectively, written $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$. The composition of two substitutions R and S is denoted by $S \circ R$. The type expression resulting from the application of a substitution S to the type variables of a type expression σ is denoted by $S\sigma$. The coercion set resulting from the application of a substitution S to the type variables of a coercion set C is denoted by $S\mathrm{C}$ i.e.

$$S\mathrm{C} \equiv \{S\sigma < S\tau \mid \sigma < \tau \in \mathrm{C}\}$$

The *unification* [Rob65] of two type expressions σ and τ is denoted by $\mathrm{UNIFY}(\sigma = \tau)$ and returns a most general unifying substitution for σ and τ if one exists, or else fails. More generally, if P is a set of type equations $\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$, then a substitution S *unifies* P if $S\sigma = S\tau$ for all $\sigma = \tau \in \mathrm{P}$, and the unification of P is denoted by UNIFY(P). Unification is extended to type environments by using the notation $\mathrm{A}_1 = \mathrm{A}_2$ for the following set of type equations:

$$\mathrm{A}_1 = \mathrm{A}_2 \equiv \{\sigma = \tau \mid x :: \sigma \in \mathrm{A}_1 \text{ and } x :: \tau \in \mathrm{A}_2\}$$

Two type expressions σ and τ *match* if they differ only in occurrences of type variables and atomic types. Given a coercion $\sigma < \tau$ such that σ and τ match, a semantically equivalent *atomic* coercion set can be computed using the algorithm ATOMIC below. This is as given in [Mit91] but extended to cater for structured types in our case:

$$\begin{array}{ll}
\mathrm{ATOMIC}(\iota < \kappa) & = \{\iota < \kappa\} \\
\mathrm{ATOMIC}(\alpha < \kappa) & = \{\alpha < \kappa\} \\
\mathrm{ATOMIC}(\iota < \beta) & = \{\iota < \beta\} \\
\mathrm{ATOMIC}(\alpha < \beta) & = \{\alpha < \beta\} \\
\mathrm{ATOMIC}(\sigma_1 \to \sigma_2 < \tau_1 \to \tau_2) & = \mathrm{ATOMIC}(\tau_1 < \sigma_1) \cup \mathrm{ATOMIC}(\sigma_2 < \tau_2) \\
\mathrm{ATOMIC}(\kappa^n \sigma_1 \ldots \sigma_n < \kappa^n \tau_1 \ldots \tau_n) & = \bigcup_{i=1}^{n} \mathrm{ATOMIC}(\sigma_i < \tau_i)
\end{array}$$

Given a substitution S and a coercion $\sigma < \tau$ such that $S\sigma$ and $S\tau$ match,

$$S \cdot \sigma < \tau \equiv \mathrm{ATOMIC}(S\sigma < S\tau)$$

A substitution S is said to *respect* a coercion set C if for all $\sigma < \tau \in \mathrm{C}$, $S\sigma$ and $S\tau$ match. If S respects C, the $\cdot$ notation is extended as follows:

$$S \cdot \mathrm{C} \equiv \{S \cdot \sigma < \tau \mid \sigma < \tau \in \mathrm{C}\}$$

4.2 The $\mathcal{M}$ algorithm

Our type inference algorithm, $\mathcal{M}$, takes a type environment, an atomic coercion set, and an expression E, and returns a triple consisting of a new type environment containing typing statements for any free and let-bound variables in E, an associated atomic coercion set, and a type expression for E. The initial coercion set, C_0, and type environment, A_0, passed to $\mathcal{M}$ are defined as follows.

Let $c_1, \ldots, c_n$ be the names of the built-in functions, current value constructors and current data functions. Let $(\mathrm{C}_1, \tau_1), \ldots, (\mathrm{C}_n, \tau_n)$ be their respective (Mitchell) types, and assume that no type variable appears in more than one of these n types. Then $\mathrm{A}_0 = \{c_1 :: \tau_1, \ldots, c_n :: \tau_n\}$ and $\mathrm{C}_0 = \mathrm{C}_1 \cup \ldots \cup \mathrm{C}_n$.

As in ML, the presence of "let" and "letrec" expressions allows a form of parametric polymorphism where universal quantification of type variables is permitted at the top level of a type expression. In our definition of $\mathcal{M}$ below, the type of a polymorphic function is expressed as $\forall\alpha_1 \ldots \alpha_n.\sigma$ where $\alpha_1 \ldots \alpha_n$ are called *generic* type variables. In order to use a polymorphic function, it is necessary to *specialise* its type i.e. to substitute a type expression for each generic type variable $\alpha_1 \ldots \alpha_n$ in σ. Constructors and built-in functions are also polymorphic i.e. in the initial type environment A_0, all typing statements $c :: \tau$ are actually assumed to be $c :: \forall\alpha_1 \ldots \alpha_n.\tau$ where $n \geq 0$ and $\alpha_1, ..., \alpha_n$ are the type variables appearing in τ.

We are now ready to define the type inference algorithm, $\mathcal{M}$, considering in turn each category of expressions in the kernel syntax.

The first case for $\mathcal{M}$ types occurrences of constants i.e. constructors, data functions and built-in functions. The type of the constant c is retrieved from the type environment A and any generic type variables are replaced by fresh type variables so that the type is unique for this occurrence of c:

$\mathcal{M}$ (C,A,c)
  = let $\forall\alpha_1 \ldots \alpha_n.\sigma = \mathrm{A}(c)$
   in $([\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\mathrm{C}, \{\}, [\beta_1/\alpha_1 \ldots \beta_n/\alpha_n]\sigma)$
   where $\beta_1 \ldots \beta_n$ are fresh type variables

The next case for $\mathcal{M}$ types occurrences of variables. If a typing for the variable x exists in the type environment A (as a result of x having been introduced by a "let" or "letrec" clause), it is retrieved and any generic type variables are replaced by fresh type variables in order to create a unique type for this occurrence of x. If no typing for x exists in A (i.e. x is a free or λ-bound variable) a new type and coercion set are created for x from fresh type variables:

$\mathcal{M}$ (C,A,x)

$= ([\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\mathrm{C}, \{\}, [\beta_1/\alpha_1 \ldots \beta_n/\alpha_n]\sigma)$, if $x :: \forall\alpha_1 \ldots \alpha_n.\sigma \in \mathrm{A}$

$= (\mathrm{C} \cup \{\alpha < \beta\}, \{x :: \alpha\}, \beta)$, otherwise

where $\alpha, \beta, \beta_1 \ldots \beta_n$ are fresh type variables

Applications, $\mathrm{E}_1\mathrm{E}_2$, are typed as in [Mit91]. As Mitchell discusses, the substitution that unifies the types inferred for E_1 and E_2 (R below) may not respect their associated coercion sets (C_1 and C_2 below). Thus, it is necessary to perform a second process of unification over the set of coercions $R\mathrm{C}_1 \cup R\mathrm{C}_2$ which will find a most general substitution that respects this set, if one exists, and which will fail otherwise. This is performed by the algorithm MATCH described in [Mit84], [Mit91]. MATCH is similar to UNIFY except that type expressions are unified only up to their structure. The complexity of MATCH is the same as that of as that of UNIFY, for which a linear implementation exists [PW78]:

$\mathcal{M}$ (C,A,$\mathrm{E}_1\mathrm{E}_2$)

= let $(\mathrm{C}_1,\mathrm{A}_1,\sigma) = \mathcal{M}(\mathrm{C},\mathrm{A},\mathrm{E}_1)$

$(\mathrm{C}_2,\mathrm{A}_2,\tau) = \mathcal{M}(\mathrm{C},\mathrm{A},\mathrm{E}_2)$

$R = \mathrm{UNIFY}(\{\sigma = \tau \to \alpha\} \cup \mathrm{A}_1 = \mathrm{A}_2)$

$S = \mathrm{MATCH}(R\mathrm{C}_1 \cup R\mathrm{C}_2) \circ R$

in

$(S \cdot (\mathrm{C}_1 \cup \mathrm{C}_2), S\mathrm{A}_1 \cup S\mathrm{A}_2, S\alpha)$

where α is a fresh type variable

Ordinary λ-abstractions, $\lambda x.\mathrm{E}$, are also typed as in [Mit91] by first typing the body E and then for the argument type using the type inferred for x if x is appears free in E, otherwise using a new type variable:

$\mathcal{M}$ (C,A,λx.E)

$= (\mathrm{C}_1, \mathrm{A}_1 - \{x\}, \sigma \to \tau)$, if $x :: \sigma \in \mathrm{A}_1$

$= (\mathrm{C}_1, \mathrm{A}_1, \alpha \to \tau)$, otherwise

where $(\mathrm{C}_1,\mathrm{A}_1,\tau) = \mathcal{M}(\mathrm{C},\mathrm{A},\mathrm{E})$ and α is a fresh type variable

λ-abstractions whose argument is annotated with an atomic type, $\lambda x :: \kappa.\mathrm{E}$, are typed by independently typing $x :: \kappa$ and E. Unification and matching then bring the two typings into line with each other:

$\mathcal{M}$ (C,A,$\lambda x :: \kappa$.E)

= let $(\mathrm{C}_1,\mathrm{A}_1,\tau) = \mathcal{M}(\mathrm{C},\mathrm{A},\mathrm{E})$

$(\mathrm{C}_2,\mathrm{A}_2) = (\{\alpha < \kappa\}, \{x :: \alpha\})$

$R = \mathrm{UNIFY}(\mathrm{A}_1 = \mathrm{A}_2)$

$S = \mathrm{MATCH}(R\mathrm{C}_1 \cup R\mathrm{C}_2) \circ R$

in

$(S \cdot (\mathrm{C}_1 \cup \mathrm{C}_2), S(\mathrm{A}_1 - \{x\}), S(\alpha \to \tau))$

where α is a fresh type variable

For expressions of the form *let* x = E_1 *in* E, we first infer a type for E_1 and then infer a type for E with respect to a type environment updated with the typing for x. The process is similar to typing an application $(\lambda x.\mathrm{E})\mathrm{E}_1$ except that a let-bound variable x can have a different specialisation of its type at each occurrence in E whereas a λ-bound x cannot:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{M}\ (\mathrm{C},\mathrm{A},let\ x = \mathrm{E}_1\ in\ \mathrm{E})} \\
\quad = \text{let} & & (\mathrm{C}_1,\mathrm{A}_1,\sigma) = \mathcal{M}(\mathrm{C},\mathrm{A},\mathrm{E}_1) \\
& & (\mathrm{C}_2,\mathrm{A}_2,\tau) = \mathcal{M}(\mathrm{C}_1,\mathrm{A}+\{x :: \forall\bar{\alpha}.\sigma\},\mathrm{E}) \\
& & R = \mathrm{UNIFY}(\mathrm{A}_1 = \mathrm{A}_2) \\
& & S = \mathrm{MATCH}(R\mathrm{C}_1 \cup R\mathrm{C}_2) \circ R \\
\quad\ \ \text{in} & & \\
& & (S \cdot (\mathrm{C}_1 \cup \mathrm{C}_2), S\mathrm{A}_1 \cup S\mathrm{A}_2, S\tau) \\
\quad\ \ \text{where} & \multicolumn{2}{l}{\bar{\alpha} \text{ is the list of type variables in } \sigma \text{ which do not appear in } \mathrm{A}_1}
\end{array}
$$

Typing expressions of the form *letrec* $(x_1, \ldots, x_n) = (\mathrm{E}_1, \ldots, \mathrm{E}_n)$ *in* E is complicated by the need to unify the types of recursive expressions. We first infer types for $\mathrm{E}_1, \ldots, \mathrm{E}_n$. We then unify the type of each E_i with the type inferred for x_i in the returned type environment A_i. In addition, since any of $\mathrm{E}_1, \ldots, \mathrm{E}_n$ may reference x_i (due to mutual recursion), we need to also unify the type for x_i in their returned type environments. We achieve this by unifying over the type environments $\mathrm{A}_1, \ldots, \mathrm{A}_n$. As usual, unification is followed by matching, to give a substitution S that respects all the coercion sets. We are now in a position to type E with respect to the updated coercion set and type environment, obtaining the typing $(\mathrm{C}',\mathrm{A}',\tau)$ below. Finally, as for "let" expressions, we need to unify and match again for free variables occurring in the whole "letrec" expression:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{M}\ (\mathrm{C},\mathrm{A},letrec\ \ (x_1, \ldots, x_n) = (\mathrm{E}_1, \ldots, \mathrm{E}_n)\ \ in\ \ \mathrm{E})} \\
\quad = \text{let} & & (\mathrm{C}_i,\mathrm{A}_i,\sigma_i) = \mathcal{M}(\mathrm{C},\mathrm{A},\mathrm{E}_i) \quad \text{for all } 1 \leq i \leq n \\
& & \tau_i = \tau_i^{'}, \quad \text{if } x :: \tau_i^{'} \in \mathrm{A}_i \\
& & \quad\ = \beta_i, \quad \text{otherwise} \qquad \text{for all } 1 \leq i \leq n \\
& & R = \mathrm{UNIFY}(\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n, \mathrm{A}_1 = \ldots = \mathrm{A}_n) \\
& & S = \mathrm{MATCH}(R\mathrm{C}_1 \cup \ldots \cup R\mathrm{C}_n) \circ R \\
& & (\mathrm{C}^{'},\mathrm{A}^{'},\tau) = \mathcal{M}(S \cdot (\mathrm{C} \cup \mathrm{C}_1 \cup \ldots \cup \mathrm{C}_n), \\
& & \qquad\qquad S\mathrm{A}+\{x_1 :: \forall\bar{\alpha}_1.S\sigma_1, \ldots, x_n :: \forall\bar{\alpha}_n.S\sigma_n\},\mathrm{E}) \\
& & T = \mathrm{UNIFY}(\{S\mathrm{A}_1 \cup \ldots \cup S\mathrm{A}_n = \mathrm{A}^{'}\}) \circ S \\
& & U = \mathrm{MATCH}(T \cdot (\mathrm{C}_1 \cup \ldots \cup \mathrm{C}_n) \cup T\mathrm{C}^{'}) \circ T \\
\quad\ \ \text{in} & & \\
& & (U \cdot (\mathrm{C}_1 \cup \ldots \cup \mathrm{C}_n \cup \mathrm{C}^{'}), U\mathrm{A}_1 \cup \ldots \cup U\mathrm{A}_n \cup U\mathrm{A}^{'}, U\tau) \\
\quad\ \ \text{where} & \multicolumn{2}{l}{\bar{\alpha}_i \text{ is a list of the free variables in } S\sigma_i \text{ which do not appear in}} \\
& & S\mathrm{A}_1 \cup \ldots \cup S\mathrm{A}_n \text{ and } \beta_1, \ldots, \beta_n \text{ are fresh type variables}
\end{array}
$$

Lastly, expressions of the form *method* $\mathrm{E}_1, \ldots, \mathrm{E}_n$ are typed by first inferring a type for each E_i independently, obtaining the typing (C_i,A_i,σ_i). These typings are then unified and matched, giving the typing ($S \cdot (\mathrm{C}_1 \cup \ldots \cup \mathrm{C}_n)$, $S\mathrm{A}_1 \cup \ldots \cup S\mathrm{A}_n$,$S\alpha$) below. The algorithm LUB is next applied to the coercion set $S \cdot (\mathrm{C}_1 \cup \ldots \cup \mathrm{C}_n)$. For all type variables β that appear in $S\alpha$ and correspond to an annotated variable in one or more of $\mathrm{E}_1, \ldots, \mathrm{E}_n$, LUB replaces a set of coercions of the form $\beta < \kappa_1, \ldots, \beta < \kappa_r$ by a single coercion $\beta < \kappa$, where κ is the least upper bound of $\kappa_1, \ldots, \kappa_r$. If the least upper bound does not exist, or is not unique, then LUB fails. For example, the argument type of a function defined by two equations $f\ x :: Employee == \mathrm{E}', f\ x :: Student == \mathrm{E}''$ is $a < Person$ whereas a function defined by two equations $f\ x :: Employee == \mathrm{E}', f\ x :: Num == \mathrm{E}''$ is not well typed.

$$
\begin{array}{l}
\mathcal{M}\ (\mathrm{C},\mathrm{A},method\ \mathrm{E}_1,\ldots,\mathrm{E}_n) \\
\quad = \text{let} \quad (\mathrm{C}_i,\mathrm{A}_i,\sigma_i) = \mathcal{M}(\mathrm{C},\mathrm{A},\mathrm{E}_i) \quad \text{for all } 1 \leq i \leq n \\
\qquad\qquad R = \text{UNIFY}\ (\{\alpha = \sigma_1 = \ldots = \sigma_n\} \cup \mathrm{A}_1 = \ldots = \mathrm{A}_n) \\
\qquad\qquad S = \text{MATCH}\ (R\mathrm{C}_1 \cup \ldots \cup R\mathrm{C}_n) \circ R \\
\qquad \text{in} \\
\qquad\qquad (\text{LUB}\ (S \cdot (\mathrm{C}_1 \cup \ldots \cup \mathrm{C}_n)),\ S\mathrm{A}_1 \cup \ldots \cup S\mathrm{A}_n, S\alpha) \\
\qquad \text{where } \alpha \text{ is a fresh type variable}
\end{array}
$$

4.3 The COERCE_CHECK algorithm

It is possible that $\mathcal{M}$ succeeds but that the coercion set it returns is inconsistent with respect to the inheritance graph. For example, given a predeclared subtype relationship $Int < Real$, the coercion set $\{Real < \alpha, \alpha < Int\}$ is inconsistent. Following Fuh and Mishra [FM89], we therefore verify for each new coercion set C inferred during $\mathcal{M}$ that every subtype relationship derivable from C is also derivable from the inheritance graph. In addition to this check, we verify that each type variable in C can be substituted by an atomic type and still yield a consistent coercion set. This means that for inheritance graphs which are not complete lattices, pairs of coercions $\alpha < \iota, \alpha < \kappa$ such that ι and κ have no common subtype are considered invalid, as are pairs of coercions $\iota < \alpha, \kappa < \alpha$ such that ι and κ have no common supertype. The overall verification of a coercion set C is thus captured by the following recursive algorithm, where G is the predeclared inheritance graph:

$$
\begin{array}{l}
\text{COERCE\_CHECK(C)} = \\
\quad \forall \iota, \kappa \in AtomicTypes.\ \mathrm{C} \vdash \iota < \kappa \ \Rightarrow\ G \vdash \iota < \kappa \\
\quad \text{and} \\
\quad \forall \alpha \in TypeVars(\mathrm{C}).\ \exists \kappa \in AtomicTypes.\text{COERCE\_CHECK}([\kappa/\alpha]\mathrm{C})
\end{array}
$$

4.4 Semantics of $\boldsymbol{\lambda x :: \kappa}$.E

When during the evaluation of a query an expression of the form $\lambda x :: \kappa.\mathrm{E}$ is applied to an expression E_1, the application $(\lambda x :: \kappa.\mathrm{E})\mathrm{E}_1$ rewrites to the following expression:

$$
\begin{array}{ll}
\text{let} & (\mathrm{C},\mathrm{A},\tau) = \mathcal{M}(\mathrm{C}_0,\mathrm{A}_0,\mathrm{E}_1) \\
\text{in} & \text{if } \mathrm{C} \vdash \tau < \kappa \\
 & \text{then } (\lambda x.\mathrm{E})\mathrm{E}_1 \\
 & \text{else } \ \mathit{Fail}
\end{array}
$$

Notice that $\mathcal{M}$ cannot fail at this stage, since the overall query has already been type-checked.

5 Related work

Type inference for the λ calculus with subtypes was first studied by Mitchell [Mit84], with later work in [Mit91]. Two algorithms, $\mathcal{G}$ and $\mathcal{GA}$, are described in [Mit91]: $\mathcal{G}$ performs type inference in the presence of unrestricted subtype relationships while $\mathcal{GA}$ performs type inference in the presence of subtype relationships between atomic types only. $\mathcal{G}$ is able to type *all* expressions in the pure λ-calculus since it permits subtype relationships such as $\alpha < \alpha \to \alpha$. $\mathcal{GA}$ however types only those expressions which possess a Curry type. Our $\mathcal{M}$ algorithm is an extension of $\mathcal{GA}$ to perform type inference for the kernel language of Section 3.6.

Further work on type inference with subtypes appears in [FM89], [FM90] which give heuristics for the simplification of a coercion set and an algorithm to check the consistency of a coercion set. We have extended this in our own COERCE_CHECK algorithm to detect cases where a coercion set is unsatisfiable with respect to the inheritance graph.

Lincoln and Mitchell have analysed the complexity of Mitchell's type inference algorithms in [LM92]. Their main result is that the problem of deciding whether a term has a type, given a set of subtyping relations between atomic types, has a lower bound in NP. Moreover, they report that since the size of a most general typing for a term may be exponentially larger than the term itself, any type inference algorithm which prints the type must take exponential time.

Cardelli has used subtyping over record types to provide a semantics and type inference algorithm in a simple applicative language with classes, objects and (multiple) inheritance [Car88]. Classes are represented by record types, with possibly functional components to represent methods. Objects are represented by instances of such types. Although a formal equivalence has yet to be defined, it is easy to see that our approach possesses similar expressiveness. We can define atomic types whose constructors act as object identifiers and can appear within data functions. The values associated with these constructors correspond to the data components of Cardelli's objects. Functions over atomic types correspond to the functional components of Cardelli's objects, in our case abstracted out of the object as global functions. One advantage of our approach is that functions over structured types and multiple arguments can be inherited and overloaded in just the same way as functions over atomic types. Also, the declaration of subtype relationships between atomic types seems more in tune with PFL's semantic data model than Cardelli's structural approach to subtyping.

Cardelli and Wegner have also formalised subtyping as *inclusion polymorphism* [CW85]. Independently, Girard [Gir71] formulated the second-order λ-calculus, or System F, in which polymorphism is treated by creating parameter bindings for type variables and passing type expressions as arguments in order to specialise a function to a specific type. This form of polymorphism is known as *universal quantification.* Inclusion polymorphism arises as a special case, in the form of *bounded quantification*, in which the type variable declared in a parameter binding is constrained to be a subtype of a type. This work has been extended to deal with recursive records [CCH+89], to integrate existential types with bounded quantification in order to support both encapsulation and (record-based) inheritance [Ghe90], and to support method overloading [Cas93].

An alternative record-based approach to inheritance is that of Jategaonkar and Mitchell [JM88] which extends the language of [Mit84] with functions that allow pattern-matching over record arguments. Types are inferred for record patterns by *reversing* the coercions associated with identifiers: this is the same effect that our *unpack_c* functions have on the type of the constructor c in Section 3.3. The precise relationship between Mitchell's approach to subtyping and F-bounded quantification is an open issue, although superficially the types of the former are more general than those of the latter since more than one subtype assertion can be associated with a type variable.

6 Concluding Remarks

We have shown how the functional database language PFL can be extended to support inheritance and overloading of functions. The resulting language has a single semantic and operational framework and does not suffer from "impedance mismatch" problems. Our approach is easily applied to other functional database languages.

The specific contributions of this paper are firstly an extension of Mitchell's $\mathcal{GA}$ algorithm [Mit91] to infer types for constructors, "let" and "letrec" expressions, λ-abstractions whose bound variable is annotated with an atomic type, and overloaded functions; and secondly an extension of the best-fit pattern-matching of [FH88] to cater for overloaded functions.

With an empty inheritance graph, our $\mathcal{M}$ algorithm types precisely those expressions that are typable by the original W algorithm and our pattern-matching algorithm reduces to ordinary best-fit pattern-matching. Thus, the new version of PFL is upwards compatible with the old, and existing PFL programs need not be rewritten.

There are several directions for further work. On the implementation side, experience has shown that the size of the coercion sets can be large even for simple functions. We are thus developing heuristics to simplify the coercion sets inferred at each step of the algorithm. On the theoretical side, Mitchell has shown the soundness of the $\mathcal{GA}$ algorithm and it remains to formally show the soundness of our extensions to it. Finally, on the language side we are investigating further extensions to the type system in order to support abstraction, encapsulation and modularisation.

Acknowledgements

This work has been carried out under EPSRC grant no. GR/J 48818. We are grateful for many fruitful discussions with our collaborators on this project, Carol Small, Swarup Reddi, David Sutton, Ben Heydecker and Jiashu Wu, and also to Spyros Soukeras for his comments on this paper.

References

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Cas93] G. Castagna. $F_{\leq}^{\&}$: integrating parametric and "ad hoc" second order polymorphism. In *Proc. 4th International Workshop on Database Programming Languages (DBPL-4), New York*, pages 338–358. Springer-Verlag, 1993.

[CCH+89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. 1989 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

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

[DOO] DOOD. *Proceedings of the International Conferences on Deductive and Object-Oriented Databases.* 1989 (Kyoto), 1991 (Munich), 1993 (Phoenix, Arizona).

[FH88] A.J. Field and P.G. Harrison. *Functional Programming.* Addison Wesley, 1988.

[FM89] Y.-C. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT'89 - Proc. International Joint Conference on Theory and Practice of Software Development*, pages 167–183, 1989. Springer-Verlag LNCS 352.

[FM90] Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 70:155–175, 1990.

[Ghe90] G. Ghelli. Modelling features of object-oriented languages in second order functional languages with subtypes. Technical Report FIDE/90/3, Dept. of Computing Science, Glasgow University, 1990.

[Gir71] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'èlimination des coupres dans l'analyse et la thèorie des types. In J. E. Fenstad, editor, *2nd. Scandanavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[HK87] R. Hull and R. King. Semantic database modelling : Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[JM88] L.A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proc. 1988 ACM Conf. on LISP and Functional Programming*, pages 198–211, 1988.

[LM92] P. Lincoln and J. C. Mitchell. Algorithmic aspects of type inference with subtypes. In *19th ACM Symposium on Principles of Programming Languages*, pages 293–304, 1992.

[Mit84] J. C. Mitchell. Coercion and type inference (summary). In *Proc. 11th ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984.

[Mit91] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[PJ87] S. Peyton-Jones. *The Implementation of Functional Programming Languages.* Morgan Kaufmann, 1987.

[Pou92] A. Poulovassilis. The implementation of FDL, a functional database language. *The Computer Journal*, 35(2):119–128, 1992.

[PS91] A. Poulovassilis and C. Small. A functional programming approach to deductive databases. In *Proc. 17th International Conference on Very Large Data Bases, (VLDB 91), Barcelona*, pages 491–500, 1991.

[PS94] A. Poulovassilis and C. Small. Investigation of algebraic query optimisation for database programming languages. In *Proc. 20th International Conference on Very Large Data Bases (VLDB 94) Santiago, Chile*, pages 415–426, 1994.

[PW78] Paterson and Wegman. Linear unification. *JCSS*, pages 158–167, 1978.

[Rob65] J. A. Robinson. A machine orientated logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[SP91] C. Small and A. Poulovassilis. An overview of PFL. In *Proc. 3rd International Workshop on Database Programming Languages (DBPL-3), Nafplion*, pages 96–110, 1991.

[SS95] D. Sutton and C. Small. Extending functional database languages to update completeness. In *Proc. BNCOD-13 (these proceedings), Manchester*, July 1995.