1990

A Functional Language with Classes

Mike Veaven

Ryan Stanisfer

Dan Wetklow

Report Number:
90-946

A Functional Language with Classes

Mike Beaven Ryan Stansifer Dan Wetklow

CSD-TR-946
January 1990

1 Introduction

Recent advances in type reconstruction algorithms for type systems with subtyping hold out the possibility of a strongly-typed, object-oriented language in which type declarations are redundant. We propose a language that includes an object construct, which, following Cardelli, we model as a typed record. The methods of the object are modeled as fields with functional types. The advantage of this approach is in the elimination of the "method not found" run-time error, in addition to other benefits of strong typing. The elegance of the language ML [Mil84], which has an expressive type system and the property that all types can be inferred at compile-time, motivates us to ensure that there is a type reconstruction algorithm for our language.

We propose a design of a typed, functional language that supports objects, classes, multiple inheritance and parametric polymorphism. We give the syntax of the language, a type-checking algorithm, and semantics using reduction rules. A prototype implementation is under development. We discuss single and multiple inheritance, and the resolution of method conflicts in multiple inheritance. We also discuss various constructs of the language and how they affect type reconstruction. We give examples of programming with classes to illustrate the different flavor of typed object-oriented languages, and some of the subtleties.

In [Car84], Cardelli showed that it is possible to model objects and inheritance with records and subtyping respectively. Records can account for all of the basic features of objects, a method of an object being modeled as a record field with functional type. Inheritance was modeled by expanding the fields of a record. Wand [Wan89] gave a concrete modeling of objects, classes and inheritance in terms of the typed lambda calculus augmented with records. Objects were modeled with records and classes were modeled with functions that mapped a set of instance variables to an object.

The type of a record consists of information about its fields, specifically which fields are present and the type of each of these fields. Since we can statically determine which fields must be present in a record, we can determine the set of messages supported by an object. This means that the error message *method not found* will never appear during the execution of a type correct program, since this would be an error in field selection.

2 Syntax

We first give the syntax of the language we wish to consider in this paper. The syntax is given in tabular form. Key words are in typewriter font, while slanted font is used for syntactic categories such as identifiers. We use $\emptyset$ to denote the empty sequence.

Value Bindings	Expressions
valbind ⇒	*expr* ⇒
id = *expr*	*id*
`rec` *id* = `fn` *id* `=>` *expr*	*con*
valbind `and` *valbind*	`self`
Types	(*expr*)
type ⇒	*expr* . *id*
basetype	*expr* : *type*
typevariable	*expr expr*
type `->` *type*	`let val` *valbind* `in` *expr* `end`
`object` { *extension*	`if` *expr* `then` *expr* `else` *expr*
field ⇒	`fn` *id* `=>` *expr*
`Absent`	`class` *id* `methods` *valbind* `end`
type	`class` *id* `inherits` *inherit* `end`
extension ⇒	`class` *id* `inherits` *inherit* `methods` *valbind* `end`
id : *field* , *extension*	*inherit* ⇒
} `Empty`	*id* `from` *id* `with` *expr*
} *rowvariable*	*inherit* `and` *inherit*
Declarations	
decl ⇒	
`val` *valbind*	
decl ; *decl*	

For the sake of discussion, we present a fairly simple language without lists or functions with multiple arguments. Furthermore, the `let val` expression is used instead of the more general `let` expression, no side effects are allowed in expressions, objects may have only one instance variable, and inheritance of methods from the same object must be listed separately. All these restrictions can easily be removed, but we wish to focus on more important aspects of the `class` expression. We do not specify the entire set of *basetype* but we assume that a type `boolean` is present and we use the type `int` when convenient. Identifiers are used to denote several different things, such as value variables, type variables, row variables, method names, and so on. We have chosen disjoint

classes of identifiers for each kind of use. Each class is distinguished by the initial character as specified in the following table.

Class of Identifier	Starting Character
Value Variable	(alphabetic character)
Type Variable	'
Row Variable	&
Method Names	#

3 Evaluation

In this section, we describe the evaluation of expressions. We present a set of functions defining the values represented by each syntactic structure. These functions utilize an "environment mapping" $env : id \rightarrow values$. The environment mappings are denoted by an association list, $\{(id_1, value_1), \ldots, (id_n, value_n)\}$. We denote the empty mapping by $\emptyset$, and the union of two mappings, by $env_1 \oplus env_2$. In the case of duplication, the definitions of the second mapping override those of the first. Most of the definitions are standard, however one might wish to note that functions are given by λ−expressions and recursive functions are defined in terms of the fixed point combinator Y. The evaluation of the `class` expression returns a function which maps the instance variable of the class to an association list which is equivalent to an environment mapping.

3.1 Value Bindings

- $EvalVB(id = expr, env) = \{(id, EvalExpr(expr, env))\}$
  The evaluation of an identifier binding returns a singleton environment which consists of an association between the identifier and the evaluated expression.

- $EvalVB(valbind_1 \text{ and } valbind_2, env) = EvalVB(valbind_1, env) \oplus EvalVB(valbind_2, env)$
  Multiple value bindings are all evaluated in the same environment, the union of the mappings is returned.

- $EvalVB(\texttt{rec } id_1 = \texttt{fn } id_2 \texttt{=>} expr, env) =$
  $\{(id_1, \mathbf{Y}\lambda v \lambda x(EvalExpr(expr, env \oplus \{(id_1, v), (id_2, x)\})))\}$
  The fixed point of the given recursive function definition is associated with the first identifier and returned in a singleton environment.

3.2 Declarations

- $EvalDecl(\texttt{val } valbind, env) = EvalVB(valbind, env) \oplus env$
  The assignment statement updates the current environment with the new bindings for the identifiers which have been defined.

- $EvalDecl(decl_1; decl_2, env) = EvalDecl(decl_2, EvalDecl(decl_1, env))$
  The evaluation of the second declaration in the sequence depends on the environmental changes of the first declaration. All the modifications of both of the declarations will appear in the environment that is returned.

3.3 Expressions

- $EvalExpr(id, env) = env(id)$
  $EvalExpr(\texttt{self}, env) = env(\texttt{self})$
  The evaluation of identifiers and `self` consists of applying the environment function to the expression. An error, which we do not specify, occurs if the identifier or `self` is undefined in the environment.

- $EvalExpr(expr_1\ expr_2, env) = EvalExpr(expr_1, env)\ (EvalExpr(expr_2, env))$
  Function application requires that the evaluation of the first expression yield a function, which can then be applied to the evaluation of the argument. The result of this application is returned.

- $EvalExpr((expr), env) = EvalExpr(expr, env)$
  Parentheses affect only parsing, not evaluation.

- $EvalExpr(expr : type, env) = EvalExpr(expr, env)$
  The type constraint on the expression does not affect the evaluation, it only affects the type checking phase during compilation.

- $EvalExpr(\texttt{if}\ expr_{bool}\ \texttt{then}\ expr_{true}\ \texttt{else}\ expr_{false}, env) =$
  $EvalExpr(expr_{true}, env)$ when $EvalExpr(expr_{bool}, env) = true$, but
  $EvalExpr(expr_{false}, env)$ when $EvalExpr(expr_{bool}, env) = false$
  The `if` expression evaluates the condition, which must return a boolean value, and on the basis of that value decides which of the two conditional expressions to evaluate and return.

- $EvalExpr(\texttt{let val}\ valbind\ \texttt{in}\ expr\ \texttt{end}) = EvalExpr(expr, env \oplus EvalVB(valbind, env))$
  The `let` expression evaluates the given expression in the environment augmented by the environment returned from the evaluation of the value binding.

- $EvalExpr(\texttt{fn}\ id\texttt{=>}expr) = \lambda x(EvalExpr(expr, env \oplus \{(id, x)\}))$
  Function definition returns a functional value mapping the given identifier to the given expression.

- $EvalExpr(expr.id, env) = EvalExpr(id, r)\ r$
  where r is $EvalExpr(expr, env)$
  The evaluation of *expr* must return an `object`, an association list matching method names

to method bodies. Method bodies are stored as functions from `objects` to values with the input to the function being the value of `self` for that object. The function representing the proper method as given by the identifier, *id*, is selected from the association list and applied to the `object` to define the true value of `self` in the method.

- $EvalExpr(\texttt{class}\ id\ \texttt{inherits}\ inherit\ \texttt{methods}\ valbind\ \texttt{end}, env) =$
  $\lambda x \lambda s((EvalIN(inherit, env \oplus \{(id, x)\})) \oplus (EvalMB(valbind, env \oplus \{(id, x)\})))$
  The `class` expression returns a function mapping the given identifier (class variable) to a value of type `object`. This object is an association list, identical in form with an environment. It maps method names to functions which represent method bodies. The functions take an object (`self`) and return the evaluation of the method. To create these methods, either the definitions can be given directly in a method clause or the methods can be inherited from an existing class. This `object` is defined by applying a class definition to a value for the class' instance variable. Method name conflicts are prohibited.

- $EvalExpr(\texttt{class}\ id\ \texttt{methods}\ valbind\ \texttt{end}, env) = \lambda x \lambda s(EvalMB(valbind, env \oplus \{(id, x)\}))$
  This is identical to the full `class` expression except that no inheritance clauses are present.

- $EvalExpr(\texttt{class}\ id\ \texttt{inherits}\ inherit\ \texttt{end}, env) = \lambda x \lambda s(EvalIN(inherit, env \oplus \{(id, x)\}))$
  This is identical to the full `class` expression except that no explicit method bindings are present.

3.4 Method Bindings

- $EvalMB(id = expr, env) = \{(id, \lambda z(EvalExpr(\texttt{expr}, env \oplus \{(\texttt{self}, z)\})))\}$
  The evaluation of an identifier binding returns a singleton environment which associates the identifier with a function mapping the value of `self` to the value of the expression.

- $EvalMB(valbind_1\ \texttt{and}\ valbind_2, env) = EvalMB(valbind_1, env) \oplus EvalMB(valbind_2, env)$
  Multiple value bindings are all evaluated in the same environment and the union of the mappings is returned.

- $EvalMB(\texttt{rec}\ id_1 = \texttt{fn}\ id_2\texttt{=>}expr, env) =$
  $\{(id_1, \lambda z(\mathbf{Y} \lambda v \lambda x(EvalExpr(expr, env \oplus \{(id_1, v), (id_2, x), (\texttt{self}, z)\}))))\}$
  A function mapping the value of `self` to the fixed point of the given recursive function definition is associated with the first identifier and returned in a singleton environment.

3.5 Inheritance

- $EvalIN(id\ \texttt{from}\ id'\ \texttt{with}\ expr, env) =$
  $\{(id, EvalExpr(id, EvalExpr(id', env)\ EvalExpr(expr, env)))\}$

The `with` clause instantiates the instance variable of the given class for the objects given by id'. We then return the environment mapping the label to the value of the same label in the given object.

- $EvalIN(inherit_1 \text{ and } inherit_2, env) = EvalIN(inherit_1, env) \oplus EvalIN(inherit_2, env)$
  Multiple inheritances are all evaluated and the union of the mappings is returned.

4 Examples

In this section we give some simple examples of the language. We will assume that the environment contains some predefined functions or constants such as $+, -, *, 0, 1$, and others as needed.

Defining a class of objects with an instance variable, `x`, and methods `#a` and `#b`, is accomplished by using the `class` construct.

```
val C = class x methods #a = x and #b = (fn z => x+z) end;
```

This defines a class value `C` which has the following type:

```
C: int -> object{#a:int,#b:int->int}Empty
```

The type of `C` is a function type.

Instantiation of classes is accomplished through function application. The following example creates two distinct instances of the class `C`.

```
val O = C 0;
val O' = C 1;
```

In the first case, the instance variable `x` has value 0; in the second it has 1.

A definition utilizing `self` is as follows.

```
val C' = class x methods #a = x and #b = fn z => (self.#a)+z end;
```

This has the same type and behavior as `C`. To illustrate inheritance, we create a new class `C''` which inherits methods from `C` and `C'`.

```
val C'' = class x
            inherits #a from C with x and #b from C' with 1
            methods #c = self.#a + (self.#b 3)
          end;
```

The type of the value of `C''` is

```
C'': int -> object{#a:int,#b:int->int,#c:int}Empty
```

A computationally equivalent definition of C'' that does not inherit methods from C and C' is

```
val C''' = class x
              methods #a = x
                  and #b = fn z => 1+z
                  and #c = self.#a + self.#b
           end;
```

Note that in all class definitions, any method name not explicitly appearing in the methods list or in an inheritance clause is assumed to be absent. This information is contained in the extension Empty. One can think of Empty as representing the infinite sequence

l_1 : Absent, l_2 : Absent, ...

where the labels $l_1, l_2, \ldots$ are all *other* labels. Therefore, in values of type

```
object{#a:int}Empty
```

we can infer that the object has only one method, namely #a which has type int and that all other methods are absent.

Row variables are used primarily when an object is passed as a parameter to a function, as in the following example.

```
val f = fn x => (x.#m)+3;
```

Here x is an object with a method #m of type int. The row variable shows that x may or may not have methods other than #m. We cannot infer any information about the existence, nonexistence, or type of any other methods of x. In the type of f, the row variable is denoted by &a:

```
f:object{#m:int}&a->int.
```

Next, we consider a slightly more complicated example, the class Int of objects modeling the integers. Each object has methods returning the value of the object, the objects representing the successor and predecessor of the object, and the function #Plus.

```
val Int = class x
             methods #Val  = x
                 and #Succ = Int (x+1)
                 and #Pred = Int (x-1)
                 and #Plus = fn y => if x = 0 then y
                                     else self.#Pred.#Plus y.#Succ
          end;
```

5 Type System

In this section, we define the type system of the language. In this type system, as in ML, we have types, represented with the metavariable τ, and type schemes, represented with the metavariable σ. A type scheme is a quantified type; type variables and row variables, denoted with the metavariable ρ, can be universally quantified. A quantified variable is also known as a *generic variable*. We distinguish between types and type schemes for the purpose of type reconstruction. Quantified types must be restricted in our typing rules as in [Mil78] and [DM82], otherwise the type reconstruction problem may be undecidable.

The type system, with the row and field variables, is Wand's [Wan89], which was inspired by Remy [Rem89].

5.1 Definitions and Notation

The following notation and definitions are used in the typing rules. A is a metavariable representing type assignments, which are mappings from variables to types and type schemes. For an expression to be typed, the type of each free variable must be known, as seen in typing rule (IDENT) in the following section. The type assignment is used to keep track of the types of the variables in the expressions. A row or type variable is called "free in A" if it does not appear in the range of A. We denote the type scheme obtained when all row and type variables in τ that are not free in A are quantified as $gen(A,\tau)$. For example, let A be the type assignment that maps v to β and τ be the type $\texttt{object}\{\texttt{\#a} : \alpha\texttt{->}\beta\}\rho$,then $gen(A,\tau)$ is $\forall\alpha\forall\rho.\texttt{object}\{\texttt{\#a} : \alpha\texttt{->}\beta\}\rho$. In a sense, this is the most general version of type τ that can be obtained given type assignment A. A *generic instance* of a type scheme is obtained when some of its quantified type and row variables are instantiated. Let σ be the type scheme $\forall\rho_1...\rho_n\forall\alpha_1...\alpha_m.\tau_1$. Any type scheme that can be obtained by generically instantiating some of the ρ_i's and α_i's is a generic instance of σ.

5.2 Typing Rules

We associate with every construct in the language a typing rule. The typing rules for the class construct are new, the others follow [CDDK86]. The typing rule for the class construct is broken down into two simpler cases with method definition separate from inheritance. Deriving the rule for the combined case is straightforward.

(IDENT) $\dfrac{}{A \vdash id : \tau}$ $A(id) = \sigma$ and τ is a generic instance of σ

(SELF) $\dfrac{}{A \vdash \texttt{self} : \tau}$ $A(\texttt{self}) = \sigma$ and τ is a generic instance of σ

(ABS) $\dfrac{A[id \mapsto \tau_1] \vdash e : \tau_2}{A \vdash (\texttt{fn } id \texttt{ => } e) : \tau_1\texttt{->}\tau_2}$

(APPL) $$\frac{A \vdash e_1 : \tau_1 \texttt{->} \tau_2 \quad A \vdash e_2 : \tau_1}{A \vdash (e_1\ e_2) : \tau_2}$$

(COND) $$\frac{A \vdash e_1 : \texttt{bool} \quad A \vdash e_2 : \tau \quad A \vdash e_3 : \tau}{A \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \ \texttt{else}\ e_3) : \tau}$$

(TYPED) $$\frac{A \vdash e : \tau}{A \vdash (e : \tau) : \tau}$$

(LET) $$\frac{A \vdash e' : \tau' \quad A[id \mapsto gen(A, \tau')] \vdash e : \tau}{A \vdash (\texttt{let val}\ id = e'\ \texttt{in}\ e\ \texttt{end}) : \tau}$$

(FIX) $$\frac{A[id_1 \mapsto \tau_1 \texttt{->} \tau_2][id_2 \mapsto \tau_1] \vdash e : \tau_2 \quad A[id_1 \mapsto gen(A, \tau_1 \texttt{->} \tau_2)] \vdash e' : \tau_3}{A \vdash (\texttt{let val rec}\ id_1 = \texttt{fn}\ id_2\ \texttt{=>}\ e\ \texttt{in}\ e'\ \texttt{end}) : \tau_3}$$

(SELECT) $$\frac{A \vdash e : \texttt{object}\{m : \tau\}\rho}{A \vdash (e.m) : \tau}$$

(METHOD) $$\frac{A[id \mapsto \tau][\texttt{self} \mapsto \texttt{object}\{m : \tau'\}\texttt{Empty}] \vdash e : \tau'}{A \vdash (\texttt{class}\ id\ \texttt{methods}\ \ m = e\ \texttt{end}) : \tau \texttt{->} \texttt{object}\{m : \tau'\}\texttt{Empty}}$$

(INHRT) $$\frac{\begin{array}{l} A[id_1 \mapsto \tau_1][\texttt{self} \mapsto \texttt{object}\{m : \tau\}\texttt{Empty}] \vdash e : \tau_2 \\ A[id_1 \mapsto \tau_1][\texttt{self} \mapsto \texttt{object}\{m : \tau\}\texttt{Empty}] \vdash id_2 : \tau_2 \texttt{->} \texttt{object}\{m : \tau\}\rho \end{array}}{A \vdash (\texttt{class}\ id_1\ \texttt{inherits}\ m\ \texttt{from}\ id_2\ \texttt{with}\ e\ \texttt{end}) : \tau_1 \texttt{->}\ \texttt{object}\{m : \tau\}\texttt{Empty}}$$

6 Type Reconstruction

The typing rules are designed so that it is possible to write a *type reconstruction*, or *type inference*, algorithm for the language. We wish to be able to determine the type of every expression at compile-time without declarations in which the users must explicitly give type information.

6.1 Unification

The instantiation of variables to make two terms equal is called "unification." If there is no instantiation of variables that will equate the two expressions, then unification is said to fail. Unification plays an important part in type reconstruction algorithms instantiating type variables to equate type expressions. From (APPL) it is seen that formal and actual parameters must have the same type. Typing rule (COND) specifies that both branches of a conditional must have the same type. The unification function is used to ensure that these requirements are met. A type

reconstruction algorithm for the typing rules given earlier will fail only if a call to unification fails or if there is an undeclared variable.

The instantiation of variables is returned via a *substitution*. For our purposes, a substitution is a mapping from type variables to type schemes and from row variables to field sequences and extensions. Substitutions are denoted with the metavariable S where S may also have an integer subscript.

The following theorem details the pertinent properties of the unification algorithm [Rob65]

THEOREM: There is an algorithm Unify, which when given two types, τ and τ' and a substitution S will either return a substitution, S' unifying $S\tau$ and $S\tau'$ or fail.

- If Unify(τ, τ', S) succeeds with S' then $S'\tau = S'\tau'$ and there exists a substitution T such that $S' = TS$.
- If there exists a substitution R such that $RS\tau = RS\tau'$ then Unify(τ, τ', S) will succeed with S' and there exists a substitution T such that $R = TS'$.

The unification algorithm we use is slightly different from the one generally used in that it has three parameters. Sometimes, unification algorithms do not take a substitution as a parameter. By modifying unification, we can implement the type reconstruction algorithm more efficiently by reducing the number of substitution composition operations.

6.2 Type Reconstruction Algorithm

TypeOf is a type reconstruction algorithm that takes a type environment, an expression and a substitution as input and returns a substitution and type. Usually, a type reconstruction algorithm does not take a substitution as a parameter, but as mentioned in the discussion on unification we choose to write it in this fashion for efficiency reasons. All of the substitution compositions are performed in the unification function. We use $\lfloor\sigma\rfloor$ to denote the type obtained when every quantified variable in the type scheme σ has been instantiated to a fresh type or row variable. Each case in the algorithm corresponds to a rule in the simplified type rules. The combination of the two separate cases for the class construct is not difficult.

TypeOf(A, e, S)=
case e of
  $id \Rightarrow$
    if $id \in dom(A)$ then
      Return $\langle\emptyset, \lfloor SA(id)\rfloor\rangle$
    else FAIL
  `self` $\Rightarrow$

if self $\in dom(A)$ then
Return $(\emptyset, \lfloor SA(\texttt{self}) \rfloor)$
else FAIL

fn id => e_1 $\Rightarrow$
let α be a fresh type variable
$\langle S_1, \tau \rangle \leftarrow$TypeOf$(A[id \mapsto \alpha], e_1, S)$
Return $\langle S_1, S_1\alpha \texttt{->} \tau \rangle$

e_1 e_2 $\Rightarrow$
$\langle S_1, \tau_1 \rangle \leftarrow$TypeOf$(A, e_1, S)$
$\langle S_2, \tau_2 \rangle \leftarrow$TypeOf$(A, e_2, S_1)$
let α be a fresh type variable
$S_3 \leftarrow$Unify$(\tau_1, \tau_2 \texttt{->} \alpha, S_2)$
Return$\langle S_3, S_3\alpha \rangle$

if e_1 then e_2 else e_3 $\Rightarrow$
$\langle S_1, \tau_1 \rangle \leftarrow$TypeOf$(A, e_1, S)$
$S_2 \leftarrow$Unify$(bool, \tau_1, S_1)$
$\langle S_3, \tau_2 \rangle \leftarrow$TypeOf$(A, e_2, S_2)$
$\langle S_4, \tau_3 \rangle \leftarrow$TypeOf$(A, e_3, S_3)$
$S_5 \leftarrow$Unify(τ_2, τ_3, S_4)
Return$\langle S_5, S_5\tau_3 \rangle$

let val rec id_1 = fn id_2 => e in e' end $\Rightarrow$
α and β are fresh type variables
$\langle S_1, \tau_1 \rangle \leftarrow$TypeOf$(A[id_1 \mapsto (\alpha \texttt{->} \beta)][id_2 \mapsto \alpha], e, S)$
$S_2 \leftarrow$Unify(β, τ_1, S_1)
$\langle S_3, \tau_2 \rangle \leftarrow$TypeOf$(A[id_1 \mapsto gen(S_2A, S_2(\alpha \texttt{->} \beta))], e', S_2)$
Return$\langle S_3, \tau_2 \rangle$

$(e : \tau)$
$\langle S_1, \tau_1 \rangle \leftarrow$TypeOf$(A, e, S)$
$S_2 \leftarrow$Unify(τ_1, τ, S_1)
if $S_2\tau = \tau$ then Return$\langle S_2, \tau \rangle$
else FAIL

let val id = e_1 in e_2 end
$\langle S_1, \tau_1 \rangle \leftarrow$TypeOf$(A, e_1, S)$
$\langle S_2, \tau_2 \rangle \leftarrow$TypeOf$(A[id \mapsto gen(S_1A, \tau_1)], e_2, S_1)$
Return $\langle S_2, \tau_2 \rangle$

$e.m$
α a fresh type variable
ρ a fresh row variable
$\langle S_1, \tau_1 \rangle \leftarrow$TypeOf$(A, e_1, S)$

$S_2 \leftarrow$Unify$(\tau_1, \texttt{object}\{m : \alpha\}\rho, S_1)$
Return$\langle S_2, S_2\alpha\rangle$
`class` id `methods` $m = e$ `end`
α and β fresh type variables
$\langle S_1, \tau\rangle \leftarrow$TypeOf$(A[id \mapsto \alpha][\texttt{self} \mapsto \beta], e, S)$
$S_2 \leftarrow$Unify$(\beta, \texttt{object}\{m : \tau\}\texttt{Empty}, S_1)$
Return$\langle S_2, S_2(\alpha\texttt{->}\beta)\rangle$
`class` id_1 `inherits` m `from` id_2 `with` e `end`
α, β, and γ are fresh type variables
ρ a fresh row variable
$\langle S_1, \tau_1\rangle \leftarrow$TypeOf$(A[id_1 \mapsto \alpha][\texttt{self} \mapsto \beta], e, S)$
$\langle S_2, \tau_2\rangle \leftarrow$TypeOf$(A[id_1 \mapsto \alpha][\texttt{self} \mapsto \beta], id_2, S_1)$
$S_3 \leftarrow$Unify$(\tau_2, \tau_1\texttt{->}\texttt{object}\{m : \gamma\}\rho, S_2)$
$S_4 \leftarrow$Unify$(\beta, \texttt{object}\{m : \gamma\}\texttt{Empty}, S_3)$
Return$\langle S_4, S_4(\alpha\texttt{->}\beta)\rangle$
`end`

7 Subtyping

In this section we give examples of how subtype polymorphism manifests itself in the language. The basic idea of a subtype is that τ_1 is a subtype of τ_2 if an expression of type τ_1 can be used anywhere that an expression of type τ_2 is used without causing an error. Another way of viewing it is that an expression having type τ_1 also has type τ_2. We denote the fact that τ_1 is a subtype of τ_2 by $\tau_1 \leq \tau_2$.

One way of expressing subtype polymorphism would be to add a typing rule:

(SUB) $\quad \dfrac{A \vdash e_1 : \tau'}{A \vdash e_1 : \tau} \qquad \tau' \leq \tau$

where "$\leq$" defines a subtype relation between types, or to have the type system set up so that adding this rule to the existing type rules has no effect on the typing of expressions. Typing rule (SUB) states that if an expression has type τ' and τ is a supertype of τ', then that expression is also of type τ.

Cardelli [Car84] defined a subtype relation on functions and records. In the context of the type system introduced in this paper the subtype relation is defined in terms of functions and objects. The subtype relation for objects states that object $\texttt{O}_1$ is a subtype of object $\texttt{O}_2$ if and only if every method in $\texttt{O}_2$ is also a method in $\texttt{O}_1$, and for each of these methods `#m`, the type of $\texttt{O}_1$`.#m` is a subtype of the type of $\texttt{O}_2$`.#m`. It is easily seen that an object $\texttt{O}_1$ can be used anywhere that $\texttt{O}_2$ can be used since the extra methods of $\texttt{O}_1$ can simply be ignored. The subtype relation for functions

states that a function f_1 is a subtype of a function f_2 if and only if the domain of f_2 is a subtype of the domain of f_1 and the range of f_1 is a subtype of the range of f_2.

The typing rules are similar to those in [DM82], with the modification that there is one typing rule for each construct in the language, as per [CDDK86]. There are also additional rules that allow object-oriented programming. The type system comes from [Rem89] and [Wan89]. There is one typing rule for each construct in the language. The rule for typing function applications specifies that the types of the formal and actual parameters must be the same. The rule for typing expressions of the form: `e.#m` demands that `e` be of type `object` and that it have a method `#m`. Another construct that must be mentioned is the `let` expression, which has the form: `let val v=e in e' end`. This is semantically equivalent to: `(fun v=>e') (e)`. The difference is that in the `let` expression, the type of the actual parameter `e` is known when deriving the type of `e'`. This has certain advantages in type reconstruction. In particular, this additional information allows the type of the actual parameter to be universally quantified. In other words, `v` can be used polymorphically in `e'`.

The advantage of adding subtyping to the type system is that it gives us a richer type structure. Consider a function that demands its parameters be objects with methods `#a:int->int` and `#b:bool->string`. Should we be able to apply this function to an object that has appropriate `#a` and `#b` methods and also has a method `#c` of some type? It will not present any problems with correctness in the execution of the program; the extra methods will simply be ignored by the function. However, the rule for typing function applications states that the formal and actual parameters must have the same type. If we have no notion of subtype polymorphism, then an object with two methods cannot have the same type as an object with three methods, therefore the function cannot be applied to objects with extra methods. The typing rules seem unduly restrictive in this case. If we add rule (SUB) with Cardelli's subtype relation, then the function can be applied to objects with extra methods. One way of viewing it is that we use typing rule (SUB) to coerce the actual parameter to a supertype that only has methods `#a` and `#b`, and then use the rule for typing function application.

In our system, subtyping is obtained by the use of row variables. Row variables are used to add fields to objects. The subtyping present in our type system is not as powerful as Cardelli's subtype relation. In other words, adding rule (SUB), where "$\leq$" is defined as Cardelli's subtype relation, to our typing rules would permit us to derive types for expressions that currently cannot be typed using the typing rules of our language. Consider the following expression:

```
val f = fn v => v.#m(3)+4
```

From the typing rules, we can derive that `f` has the following type:

```
object{#m:int->int}&a -> int
```

If `x` and `y` have types

```
object{#m:int->int,#l:bool->bool}Empty
object{#m:int->int,#w:str->str}Empty
```

in the type environment, then the expression

```
(fn f => f(x)+f(y))(fn v => v.#m(3)+4)
```

cannot be typed using the given rules. To type `f(x)`, the row variable `&a` must have `#l:bool->bool`, while at the same time `&a` must have `#l:Absent` to type `f(y)`. Since it cannot have both, we cannot type the expression. However, if we added typing rule (SUB) to our typing rules with Cardelli's subtype relation this expression could be typed since both `x` and `y` have a method `#m` of type `int->int`, which is really all that function `f` requires.

This does not mean that there is no subtype polymorphism present in our type system. Note that in the above example `f` has type `object{#m:int->int}&a -> int`. This means that it can be applied to any object that has a method `#m` of type `int->int`, i.e., it can be applied to all and only subtypes (using the subtype relation defined by Cardelli) of `object{#m:int->int}Empty`, since the row variable `&a` can be instantiated to include any additional methods. This is an example of subtype polymorphism.

The reason that we could not type the expression in the first example was because of the presence of both `f(x)` and `f(y)`. We could not instantiate the row variable, `&a`, two different ways in the same context. Note that if we rewrote the above expression in the computationally equivalent form:

```
let val f = fn v => v.#m(3)+4 in f(x)+f(y) end
```

then this expression can be typed with our typing rules because the row variable `&a` is universally quantified. For subtype polymorphism, as well as parametric polymorphism, the `let`-expression is required if we wish to use a parameter polymorphically in an expression. The reason for this is that we are using the same technique to obtain subtype polymorphism (row variables) as we used to obtain parametric polymorphism (type variables).

8 Object-Oriented Programming

8.1 Inheritance

There are many views of the relationship between inheritance and subtyping. Generally there are two choices: strict or nonstrict inheritance. Strict inheritance means that if `A` inherits from `B`, then `A` *is a* `B`. In other words, `A` is a subtype of `B`, and inheritance is equivalent to subtyping. This imposes a well-defined relation between a class and classes that inherit from it. Nonstrict inheritance is less restrictive. The idea is that if `A` inherits from `B`, then `A` *is like* `B`, but not necessarily *is a* `B`, since `A` can redefine methods in any way that it pleases. Both strict and nonstrict inheritance are concerned with code reuse, but strict inheritance is concerned with the organization of data as well.

8.1.1 Single, Strict vs. Single, Nonstrict Inheritance

In this discussion, we use the terminology A is a *superclass* of B when B inherits some methods from A. When we use the terms subtype and supertype we are using Cardelli's subtype relation. The choice between strict and nonstrict inheritance could be more easily decided if one was conceptually simpler than the other. The following example illustrates that, at least on a high level, there is not much difference between the two choices as far as modeling goes.

```
val P = class x
            methods #a = (fn y => if x then y else 0)
        end
val Q = class x
            inherits #a from P with x
            methods #a = (fn y => if x then y else false)
                and #b = (fn y => if x then y else 0
        end
```

Note that method name #a is defined in the method part of the declaration and is also inherited from P. If we decide that the second definition of the method is the definition that will be used by class Q, then we have nonstrict inheritance. Q inherits from P, but would not be a subtype of it, since:

```
P:bool->object{#a:int->int}Empty
Q:bool->object{#a:bool->bool,#b:int->int}Empty
```

and the #a method in the range of Q would not be a subtype of the #a method in the range of P. On the other hand, if the first definition of #a (the inherited definition) is to be the definition of method #a in Q, then we have strict inheritance. The types of P and Q would be:

```
P:bool->object{#a:int->int}Empty
Q:bool->object{#a:int->int,#b:int->int}Empty
```

Class Q inherits from P and is a subtype of P. As is easily seen, the only difference between the two models is whether or not an inherited method can be redefined (or forgotten). From the standpoint of conceptual simplicity, neither seems inherently superior to the other.

In our language we have nonstrict inheritance. The reason for this will be explained in the following discussion on single and multiple inheritance.

8.1.2 Single vs. Multiple Inheritance

Single inheritance means that each class has a unique superclass; multiple inheritance means that a class can have many superclasses, not necessarily related. Although multiple inheritance seems

more powerful, there are many problems that exist in modeling multiple inheritance that do not arise with single inheritance.

If we permit multiple inheritance, we must have some way of resolving method name conflicts. This problem was not difficult when modeling single inheritance. Since there was only one unique supertype, the problem could be reduced to a single question: Is it permissible to redefine inherited methods? This was the only sort of method name conflict possible. With multiple inheritance we can also have method name conflicts between superclasses.

The problem is more complex than simply deciding which superclasses' method should be inherited in case of a conflict. Earlier, we associated strict inheritance with subtyping and a vague *is a* relation. If `A` was a subclass of `B`, then `A` was a subtype of `B` and we had the relation `A` *is a* `B`. This was easily modeled when we had single inheritance, as the examples in the previous section show. If `A` was a subclass of `B`, then every method in `B` was a method in `A`. `A` had the same behavior as `B` for those methods and it had a few extra methods as well. It seemed reasonable to say that `A` was also a `B`. Our informal *is a* relation presents problems when we try to maintain it under multiple inheritance.

To illustrate the difficulties involved let `A` be a subclass of both `B` and `C`. Furthermore, suppose there is a method name `#m` defined in class `B` and in class `C` where `#m` is the *factorial* function in class `B` and is the *square* function in class `C`. We must decide what is the definition of method `#m` in `A`. If we arbitrarily choose a methods `#m` from one of the two superclasses, `A` is still a subtype of both `B` and `C`, but is our vague notion of an *is a* relation realized? Since the methods have such different behaviors, we argue that it is not. Our informal notion of an *is a* relation is more than a statement about type errors in the program.

We consider five possible modelings of inheritance with regard to the questions of strict or nonstrict, single or multiple inheritance.

1. Single, Strict Inheritance. `A` is a subclass of `B` implies that `A` is a subtype of `B` and that `A` *is a* `B`.

2. Single, Nonstrict Inheritance. Inheritance is a means of reusing code and has nothing to do with a subtyping which is a property of the type system. There is no notion of an *is a* relation.

3. Multiple, Strict Inheritance - option 1. Since it is undecidable in general to determine if two functions are equal, we cannot determine if the methods associated with two conflicting method names are the same. To maintain our *is a* relation we can declare that all method name conflicts are illegal.

4. Multiple, Strict Inheritance - option 2. Have some sort of internal naming scheme that allows both methods to appear in `A`. This would entail duplicating all methods that have the same name.

5. Multiple, Nonstrict Inheritance. Inheritance is a means of reusing code and has nothing to do with a subtyping which is a property of the type system. There is no notion of an *is a* relation.

Choice 3 is rejected immediately. It is unduly restrictive. Choice 4 is also rejected, it is too complex. If we decide on single inheritance, it may be preferable to have strict inheritance, deciding that the realization of our informal notion of an *is a* relation is more important than the extra flexibility obtained by allowing the programmer to redefine and omit methods inherited from a superclass. If we decide in favor of multiple inheritance, then strict inheritance becomes quite a bit more complex. For us, the choice is between single, strict inheritance and multiple, nonstrict inheritance. Since it was never obvious to us that single, strict inheritance was inherently superior to single, nonstrict inheritance we decide in favor of multiple, nonstrict inheritance.

So the final result is that subtyping and inheritance are both present, but are not equivalent in our language. Subtype relations are a property of the type system, while inheritance has nothing to do with types, but rather with code reuse.

8.1.3 Modeling Multiple Inheritance

Our modeling of multiple inheritance is as follows:

```
class x
      inherits #l from P  with Q
           and #m from P  with Q
           and #n from P' with Q'
       methods #a=M
end
```

where `P` and `P'` are classes, `x` is a parameter (or instance variable) to the class being defined, `Q` and `Q'` are parameters, or instance variables, to `P` and `P'`, `#l` and `#m` are methods in class `P`, and `#n` is a method in class `P'`. The `#a` is a method name for the class being defined, and the `M` is the associated method definition.

Method name conflicts are not a problem with this implementation since the user explicitly specifies which method will be inherited from each superclass. As a matter of fact, it is illegal to have a name conflict here. Each method name must be uniquely defined.

8.1.4 Extensions

As a shorthand, it might be desirable to have an `inherits all` capability. This would be useful if the programmer wished to inherit many methods from one superclass and a couple of other methods from other superclasses.

```
class x
    inherits #l from P with Q
         and #m from P with Q'
         and all from P' with Q
     methods #a = M
         and #b = M'
end
```

Method name conflicts would be decided in favor of the explicitly mentioned method. In other words, there can be a name conflict between a method inherited from P' under the `inherits all` and one inherited from another superclass, say P. In this case the method inherited from P takes precedence. It is also permissible to redefine a method inherited with an `inherits all`. Type reconstruction is no more difficult than before, since it is known where every method is defined.

If we allow one `inherits all`, would it present any difficulties to allow more than one? Would it be desirable to have multiple `inherits all`'s? This is even more flexible than just one `inherits all`, but a price must be paid when performing type reconstruction. Consider the situation where R inherits from P and Q with `inherits all`'s. If we then encounter the method selection expression `R.#m` and if method `#m` was not inherited explicitly from P or Q, then which superclass was method `#m` inherited from? Rules defining precedence do not help in this situation. Assume that methods from superclass P have precedence over methods from superclass Q. If `R.#m:int->int`, then either P has a method `#m` that has type `int->int` and Q might or might not have a method `#m` of some type, or P does not have any method `#m` and Q has a method `#m` of type `int->int`. This set of possibilities cannot be represented with a single type, it must be represented with three separate types. This means that there is no principal type property (i.e. the property that all possible types of any expression are instantiations of a single type in the type system), rather the principal type of an arbitrary expression will be a finite set of possible types. Although it is still possible to perform type reconstruction, it is more difficult.

Can we maintain our multiple `inherits all`'s and at the same time preserve the property that the principal type of an expression can be represented as a single type? If we decide that for any method name conflict the programmer must explicitly state from which superclass the method is being inherited, then there is not any problem. If there are no row variables in the types of the superclasses, this seems reasonable. However, it is possible to write programs where the types of the superclasses have row variables in them. For example:

```
val y = fn P => fn Q => class x
                            inherits all from P with x
                                 and all from Q with x
                        end
```

The types of classes P and Q will contain row variables in them. All that is known is that they are classes that require one instance variable. If we modify the function body so that later in it

there is the method selection expression `(y P' Q' 4).#m`, where `P'` and `Q'` are classes, then all we know is that either `P` or `Q` had method `#m` of appropriate the type. The programmer may write this function knowing that one of the actual parameters sent to this function will always have method `#m`.

We can handle this situation in several possible ways.

1. Allow multiple `inherits all`'s and the types of superclasses to have row variables. If we adopt this suggestion, then we do not have the property that all of the types derivable for an expression can be represented with a single type in the type system.

2. Allow multiple `inherits all`'s with the restriction that all row variables are considered empty. If the programmer always intends to send classes with a particular method, let him specify that in the class definition. Note that this is not quite as powerful in a practical sense as the first suggested solution. Consider where, in the example given above, the programmer always intends that one of the actual parameters have method `#m`, but not always the same parameter. In other words, sometimes `Q` will have method `#m` and other times `P` will have method `#m`. The programmer cannot specify from which superclass the method `#m` is to be inherited from when writing the class definition.

3. Only one `inherits all`. If there are any method name conflicts between an explicitly inherited method and a method inherited through an `inherits all`, then precedence is given to the explicitly inherited method.

The first suggestion is certainly the most powerful. Any program written using the second or third suggestion will work just as well using the first suggestion. The reverse of this does not hold. The price we pay for this extra flexibility is the loss of the principal type property, as we mentioned earlier.

Since the second and third suggestions both maintain the principal type property, we give examples of the strengths and weaknesses of the second and third suggestions with respect to one another.

Example 1:
Assume that we have adopted the second suggestion. Consider the following class definition

```
class x
       inherits all from P with x
            and all from Q with x
            and all from R with x
end
```

where `P`, `Q`, and `R` all have many methods and there are no name conflicts. If we had adopted the third suggestion, then writing this declaration would be much more difficult. We would have one

inherits all, and every method name in the other two superclasses would have to be mentioned explicitly. If there are some name conflicts, then the second suggestion would entail explicitly mentioning only those method names where there was a conflict. In this case, it is obvious that the second suggestion is superior to the third.

Example 2:
Assume that we have adopted the third suggestion

```
fn x => fn P => class x
                    inherits all from P with x
                    methods #m=M
                end
```

where the actual parameter corresponding with the formal parameter P is a class with many methods. The third suggestion allows us to extend P with this simple declaration while the second suggestion would require us to explicitly mention every method name present in the actual parameter, since the row variable is treated as Empty.

If the programmer is defining classes in functions so that the superclasses are often parameters to the function, the third suggested modeling is superior to the second. If the programmer generally does not define classes inside of functions where the superclasses are parameters to that function, then the second suggestion is superior to the third. The third suggestion assumes that most classes will have one superclass from which they inherit most of their methods.

While the second suggestion seems superior to the third, there is another possibility, which has the advantages of both the suggestions being considered. This is to allow multiple inherits alls and allow exactly one superclass to have a nonempty rowvariable in it. If we use inherits each instead of inherits all to signify that row variables are to be treated as Empty and inherits all to signify that method names captured by the row variable are to be inherited as well, our modeling of inheritance would look like:

```
class x
      inherits #l from P with Q
           and each from P' with Q'
           and all from P'' with Q''
       methods #a=M
end
```

where l and #a would take precedence over method inherited from P' and P'' in case of a name conflict and any method inherited from P' would take precedence over any method inherited from the row variable of P''. This gives us all the advantages of the second and third suggestions and allows us to maintain a principal type property.

Two more questions arise over this modeling. The first is whether this has gotten too complicated. Should the programmer be forced to consider the effects of row variables while writing his programs? The same question can be asked about the third suggested model. The second question is the same question asked before: Why not have multiple `inherits alls`? The answer to the second question is that this will violate the principal type property, which we are trying to maintain. To see this, note that this would be equivalent to the first suggested modeling.

In [Wan89], Wand gave a different modeling of multiple inheritance, which was follows:

class $(x_1, \ldots, x_n)$
  inherits $P(Q_1, \ldots, Q_p), P'(Q'_1, \ldots, Q'_p)$
  methods $a_1 = M_1, \ldots, a_k = M_k$
end

The basic difference between our approach and Wand's is how name conflicts among classes are handled. In Wand's approach in the case of a name conflict the second class mentioned's methods overwrite the first classes' methods. This is equivalent to inheriting from every superclass with an `inherits all` allowing row variables (ala suggestion 1), since there is no simple way of inheriting specific methods from specific superclasses in this model.

9 Examples of Type Reconstruction

In this section, examples of type reconstruction for our language are provided. Since there has already much literature on type reconstruction in the presence of parametric polymorphism the examples deal mainly with the issues of subtype polymorphism and inheritance. Our main concern is with how well the type system and typing rules in our language allow us to capture subtype polymorphism (see (SUB) in the section on subtyping). In the last part of this section we also discuss some problems that arise in type reconstruction when typing rule (SUB) is added to the type system.

9.1 Another Type System

Ideally, we would like to have a type system and typing rules such that expressions that cannot be typed would not have types if (SUB) were added to the typing rules. In other words, adding (SUB) to the typing rules would not affect the set of expressions that can be typed. Row variables would be eliminated from the type system, since they are there only to provide some subtype polymorphism and would be useless if (SUB) were added.

To see how much subtype polymorphism is present in our language, consider a type system with the following typing rules. Note that in (SELECT), $proj(\tau, m)$ denotes the type of method m in type τ. It should also be noted that the extension `Empty` is not needed in rules (METHOD) and (INHRT) since row variables have been removed from the type system, but it has been left in there for the sake of clarity in the subsequent examples.

(IDENT) $\dfrac{}{A \vdash id : \tau}$ $A(id) = \sigma$ and τ is a generic instance of σ

(SELF) $\dfrac{}{A \vdash \texttt{self} : \tau}$ $A(\texttt{self}) = \sigma$ and τ is a generic instance of σ

(ABS) $$\frac{A[id \mapsto \tau_1] \vdash e : \tau_2}{A \vdash (\texttt{fn}\ id \texttt{=>}\ e) : \tau_1 \texttt{->} \tau_2}$$

(APPL) $$\frac{A \vdash e_1 : \tau_1 \texttt{->} \tau_2 \quad A \vdash e_2 : \tau_3 \quad \tau_3 \leq \tau_1}{A \vdash (e_1\ e_2) : \tau_2}$$

(COND) $$\frac{A \vdash e_1 : \texttt{bool} \quad A \vdash e_2 : \tau_1 \quad A \vdash e_3 : \tau_2 \quad \tau_1 \leq \tau_3 \quad \tau_2 \leq \tau_3}{A \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \ \texttt{else}\ e_3) : \tau_3}$$

(TYPED) $$\frac{A \vdash e : \tau_1 \quad \tau_1 \leq \tau_2}{A \vdash (e : \tau_2) : \tau_2}$$

(LET) $$\frac{A \vdash e' : \tau' \quad A[id \mapsto gen(A, \tau')] \vdash e : \tau}{A \vdash (\texttt{let val}\ id = e'\ \texttt{in}\ e\ \texttt{end}) : \tau}$$

(FIX) $$\frac{A[id_1 \mapsto \tau_1 \texttt{->} \tau_2][id_2 \mapsto \tau_1] \vdash e : \tau_2 \quad A[id_1 \mapsto gen(A, \tau_1 \texttt{->} \tau_2)] \vdash e' : \tau_3}{A \vdash (\texttt{let val rec}\ id_1 = \texttt{fn}\ id_2\ \texttt{=>}\ e\ \texttt{in}\ e'\ \texttt{end}) : \tau_3}$$

(SELECT) $$\frac{A \vdash e : \tau_1 \quad \tau_1 \leq \texttt{object}\{m : \tau\} \quad \tau \leq proj(\tau_1, m)}{A \vdash (e.m) : \tau}$$

(METHOD) $$\frac{A[id \mapsto \tau][\texttt{self} \mapsto \texttt{object}\{m : \tau'\}\texttt{Empty}] \vdash e : \tau'}{A \vdash (\texttt{class}\ id\ \texttt{methods}\ \ m = e\ \texttt{end}) : \tau \texttt{->} \texttt{object}\{m : \tau'\}\texttt{Empty}}$$

(INHRT) $$\frac{\begin{array}{l} A[id_1 \mapsto \tau_1][\texttt{self} \mapsto \texttt{object}\{m : \tau\}\texttt{Empty}] \vdash e : \tau_2 \\ A[id_1 \mapsto \tau_1][\texttt{self} \mapsto \texttt{object}\{m : \tau\}\texttt{Empty}] \vdash id_2 : \tau_3 \texttt{->} \texttt{object}\{m : \tau\}\texttt{Empty} \quad \tau_2 \leq \tau_3 \end{array}}{A \vdash (\texttt{class}\ id_1\ \texttt{inherits}\ m\ \texttt{from}\ id_2\ \texttt{with}\ e\ \texttt{end}) : \tau_1 \texttt{->}\ \texttt{object}\{m : \tau\}\texttt{Empty}}$$

Note that there is no rule (SUB) in this system. The subtype relation is explicit in certain typing rules. This system is not equivalent to a system with rule (SUB). To see this, observe that if an expression has type `object{#a:bool}Empty`, then it also has type `object{}Empty` when (SUB) is present in the type system. This is not true for the set of typing rules we have just presented. However, our system is equivalent to a system with (SUB) in another sense. If an expression has a type under the type system with (SUB), then it is also has a type in the system we have just presented.

In this section we will study the rules that contain an explicit subtype relation to see how much subtype polymorphism is present in the type system with row variables that is used in our language. The rules that we fix our attention on are: (APPL), (COND), (TYPED), (SELECT) and (INHRT). In examining these rules we will assume that all necessary lines from the following code precedes any example given.

```
val A = class x methods #a=x end;
val B = class x methods #b=x end;
val C = class x methods #a=x and #b=x end;
val D = class x methods #a=4 and #b=x end;
```

9.2 (APPL)

In the section on subtype polymorphism, we mainly dealt with typing rule (APPL) in typing expressions of the form $e_1(e_2)$. As was shown there, row variables allow us to add methods to the domain of the function e_1 permitting e_2 to have any number of extra methods and thus providing some subtype polymorphism. This did not work if the parameter needed to be used subtype polymorphically within the body of the expression e_1. In that case, use of the semantically equivalent `let`-expression, which quantified the row variables in the type of expression e_2, was required.

9.3 (COND)

The type system used in our language is not capable of providing the same degree of subtype polymorphism in typing conditionals as it does for function applications.

```
if true then A 3 else B 3
```

Under Cardelli's subtype relation, the common supertype of both branches of the conditional is `object{}Empty`. This expression does not have a type in our language. The type of the first branch of the conditional, `object{#a:int}Empty`, cannot be unified with the type of the second branch, `object{#b:int}Empty`. It does not help to rewrite this as the following `let`-expression.

```
let
   val x = A 3 and y = B 3
in
   if true then x else y
end
```

The branches of the conditional have no row or type variables, therefore the ability to universally quantify row and type variables that the `let`-expression provides does not help in this case. The reason for the lack of success in obtaining subtype polymorphism in conditionals expressions in

our language is that we have no means of ignoring arbitrary methods in an object. In (APPL) we wished to add methods to the type of the domain of the function. Adding extra methods is the purpose of row variables in the type system in our language. Obtaining subtype polymorphism when typing conditionals often requires the forgetting of certain methods, which is not an ability that our use of row variables provides. This is not to say that there is no subtype polymorphism in our language when typing conditional expressions. The expression

```
if true then fn v=>v.#a else fn v=>v.#b
```

is type correct in our language. The type `object{#a:'a,#b:'a}&a->'a` is a supertype of both branches of the conditional. In this case, obtaining a supertype of the types of the two branches of the conditional entailed the addition instead of the deletion of methods.

9.4 (TYPED)

Typing rule (TYPED) permits the programmer to provide explicit type information to the compiler. The type reconstruction algorithm verifies that the type explicitly given by the programmer is indeed a legal type of the expression and, if so, assigns that type to the expression.

```
A 3 : object{#a:int}Empty
```

can be typed using the typing rules given in this section. `object{#a:int,#b:int}Empty` is a subtype of `object{#a:int}Empty`. However, the expression does not have a type in our language. As detailed in the discussion of (COND), our use of row variables does not permit us to forget or ignore a method. For this expression to have a type in our language, `object{#a:int,#b:int}Empty` and `object{#a:int}Empty` would have to be unifiable as seen by examining the rule

$$\frac{A \vdash e : \tau}{A \vdash (e : \tau) : \tau}$$

as it appears in the type system for our language.

As was the case in typing conditionals, there is some subtype polymorphism provided in our language in the case of explicit type information. The expression

```
fn v=>v.#a+3 : object{#a:int,#b:int}Empty->int
```

is a valid expression in our language. The programmer has demanded that this function only be applied to objects that have exactly two methods, `#a` and `#b`, both of which have type integer. This example is interesting in that it allows the programmer to demand that parameters with methods other than `#a` and `#b` are not permissible. This is not the same as ignoring a method that is already present in the type of an object. The expression

```
fn v=>v.#a+3 : object{b:int}Empty->int
```

in which the programmer is trying to forget a method already present is not a valid expression in our language.

9.5 (SELECT)

In the type system outlined in this section the expression $e.m$ has type τ if e is an object that has a method m of type τ. Our language also provides this property. The possibility of additional fields is captured with the row variable.

9.6 (INHRT)

The issues involved with subtyping and the rule (INHRT) are the same as those that were involved with typing rule (APPL).

Other issues in type reconstruction for the class construct are worth mentioning here. One is that objects created using the class construct do not have row variables in them. Note that in the typing rules

$$\text{(METHOD)} \quad \frac{A[id \mapsto \tau][\texttt{self} \mapsto \texttt{object}\{m : \tau'\}\texttt{Empty}] \vdash e : \tau'}{A \vdash (\texttt{class } id \texttt{ methods } m = e \texttt{ end}) : \tau \texttt{->object}\{m : \tau'\}\texttt{Empty}}$$

$$\text{(INHRT)} \quad \frac{\begin{array}{l} A[id_1 \mapsto \tau_1][\texttt{self} \mapsto \texttt{object}\{m : \tau\}\texttt{Empty}] \vdash e : \tau_2 \\ A[id_1 \mapsto \tau_1][\texttt{self} \mapsto \texttt{object}\{m : \tau\}\texttt{Empty}] \vdash id_2 : \tau_2 \texttt{->object}\{m : \tau\}\rho \end{array}}{A \vdash (\texttt{class } id_1 \texttt{ inherits } m \texttt{ from } id_2 \texttt{ with } e \texttt{ end}) : \tau_1 \texttt{->} \texttt{ object}\{m : \tau\}\texttt{Empty}}$$

when an object is created using the class construct, it is known what methods are present in the object. It does not matter if the types of the superclasses contain row variables. The extension Empty appears in the type of all objects created using the class construct. Row variables appear in the types of objects that are known only by the select ($e.m$) operations performed on them. This is generally seen in the body of a function. In this case, an object is not being constructed, rather a description is being constructed of the sorts of objects that will be acceptable as parameters to that function.

9.7 Additional Issues

Ignoring subtyping and inheritance for the moment, consider the expression

```
fn v => if true then v else (fn x => v)
```

which has no type in the type system used in our language. Looking at the problem from a type reconstruction viewpoint, we cannot unify α with $\beta \rightarrow \alpha$. We cannot unify α with a type τ containing α where $\tau \neq \alpha$ unless we add recursive types to the type system. This example has nothing to do with subtype polymorphism. The expression cannot be typed using the type system presented in this section either. This example involves only type variables. The same situation holds for row variables. The expression

```
fn v => if v.#a then v else B v
```

where the types of v and B v are object{#a:true}&a and object{#b:object{#a:true}&a}&b respectively, cannot be typed in our language. The types of the two branches cannot be unified. The problem is not one of forgetting fields as in the earlier examples of conditionals, rather, it is the same problem that occurs even when the only sort of polymorphism in the type system is parametric, as seen in the previous example. However, unlike the previous example, the addition of subtype polymorphism does make a difference in the typing of this expression. To see this, note that object{}Empty is a supertype of both branches of the conditional. Therefore, the expression has a type when the typing rules presented in this section are used.

An interesting example illustrating another difficulty involved in adding subtype polymorphism to the type system is

```
fn v => if true then C v else D true
```

which, since object{#a:'a,#b:'a}Empty cannot be unified with object{#a:int,#b:bool}Empty, cannot be typed in our language. 'a cannot be unified with both int and bool. Unlike the previous examples that showed only how the type system used in our language did not have as much subtype polymorphism present as would be obtained by adding (SUB) to the typing rules, this example also illustrates problems that occur with the addition of (SUB) to the type rules. object{#a:int}Empty, object{#b:bool}Empty and object{}Empty are all common supertypes of the branches of the conditional. Note that there is no one single type that can represent all of the common supertypes of the two branches. The first two supertypes are not related via Cardelli's subtype relation and neither is an instance of the other. If (SUB) is added to the typing rules or if the addition of (SUB) does not permit the typing of any new expression, then the principal type property, discussed in the section on modeling inheritance, is lost. Note that if there is an explicit (SUB) typing rule, then the principal type property generally means that for each type correct expression there exists a type such that all types for that expression are substitution instances, generic instances, and/or supertypes of that type. If there is no typing rule (SUB), rather the subtype relation is explicit in the typing rules, then the principal type property means that for every type correct expression there exists a type such that all types for that expression are substitution and/or generic instances of that type.

If we modify the expression in the previous example, supplying an integer argument

```
(fn v => if true then C v else D true) 5
```

the resulting expression has a principal type in the type system presented in this section. The type object{#a:int}Empty is the principal type of the expression. This expression still does not have a type in our language, since the conditional expression cannot be typed in our language. Even though the conditional expression has no principal type in the system presented in this section or in systems with typing rule (SUB), it still can be typed in those systems. Since the type of the

argument, integer, is constant, rewriting the expression in the semantically equivalent form of the `let`-expression will not permit it to be typed since the advantage of the `let`-expression lies in the ability to universally quantify the type and row variables in the type of the argument.

```
fn v => if true then v else v.#a
```

The above example poses an interesting problem when (SUB) is present in the typing rules. Since `v` is an object, the expression has a type only if `v.#a` is an object. This expression has a type if `v` has type

```
object{#a:object{}Empty}Empty
```

The expression also has a type if v has type

```
object{#a:object{#a:object{}Empty}Empty}Empty
```

or

```
object{#a:object{#a:object{#a:object{}Empty}Empty}Empty}Empty
```

and so on. In the previous set of examples where there was no principal type, the set of possible types of the expression could be represented by a finite set of types. That is not true in this case. Although recursive types are not needed for this expression to have a type, they are needed to express the set of possible types of this expression. The type system for our language will simply flag this expression as a type error.

References

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis(France), Springer LNCS 173*, pages 51–68, 1984.

[CDDK86] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Conf. Proc. 1986 ACM Symposium on LISP and Functional Programming*, pages 13–27, 1986.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Records of the Ninth Annual ACM Symposium of Principals of Programming Languages*, pages 207–212, 1982.

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

[Mil84] Robin Milner. Proposal for standard ML. In *Conf. Proc. 1984 ACM Symposium on LISP and Functional Programming*, pages 184–197, 1984.

[Rem89] Didier Remy. Typechecking records and variants in a natural extenstion of ML. In *Conference Records of the 16th Annual ACM Symposium of Principals of Programming Languages*, pages 77–88, 1989.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal Assoc. Comp. Mach.*, 12:23–41, 1965.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proc. Fourth IEEE Symposium on Logic in Computer Science*, pages 92–97, 1989.

```
type Id = String.string;

datatype Type =
            Int
        |   Bool
        |   TypeVariable    of Id
        |   Arrow           of Type         * Type
        |   Object          of Extension

and Field =
            Absent
        |   Present         of Type

and Extension =
            Empty
        |   RowVariable     of Id
        |   Ext             of Id           * Field         * Extension
        ;

datatype Expression =
            Identifier      of Id
        |   Self
        |   Abstraction     of Id           * Expression
        |   Application     of Expression   * Expression
        |   Constraint      of Expression   * Type
        |   Let             of ValBind      * Expression
        |   Conditional     of Expression   * Expression    * Expression
        |   Select          of Expression   * Id
        |   Class           of Id           * Inheritance   * Methods

and Methods =
            EmptyMethod
        |   Meth            of ValBind

and Inheritance =
            EmptyInheritance
        |   Inh             of Inherit

and Inherit =
            From            of Id           * Id            * Expression
        |   InhAnd          of Inherit list

and ValBind =
            ValEqn          of Id           * Expression
        |   ValRec          of Id           * Id            * Expression
        |   ValAnd          of ValBind list
        ;

datatype Declaration =
            Val             of ValBind
        |   Sequence        of Declaration list
        ;

datatype Value =
            Constant        of Id
        |   Fun             of (Value -> Value)
        |   Record          of (Id*(Value->Value)) list
        ;

type Env = (Id * Value) list;

val self = "self";

exception undefined_id;

fun LookUp(id:Id,((h1,h2)::t)) = if id = h1 then h2 else LookUp(id,t)
  | LookUp(id:Id,nil) = raise undefined_id
  ;

exception self_error;
exception syntax_error;
exception record_mismatch;
exception function_mismatch;
exception condition_mismatch;

fun ApplySelect (x as Record(mb)) id = (LookUp(id,mb)) x
  | ApplySelect (_) id = raise record_mismatch
  ;

fun InheritMethod (Record(mb)) id = [(id,LookUp(id,mb))]
  | InheritMethod (_) id = raise record_mismatch
  ;

fun EvalVB env (ValAnd(h::t)) =
        (EvalVB env (ValAnd(t)))@(EvalVB env h)
  | EvalVB env (ValAnd(nil)) =
        nil
  | EvalVB env (ValRec(id1,id2,e)) = let
        fun f x = EvalExpr e ((id2,x)::((id1,Fun(f))::env))
    in
        [(id1,Fun(f))]
    end
  | EvalVB env (ValEqn(id,e)) =
        [(id,EvalExpr e env)]

and EvalExpr (Identifier(id)) env =
        LookUp(id,env)
  | EvalExpr (Self) env =
        LookUp(self,env)
  | EvalExpr (Abstraction(id,e)) env =
        Fun(fn x => EvalExpr e ((id,x)::env))
  | EvalExpr (Application(e1,e2)) env =
        Apply (EvalExpr e1 env) (EvalExpr e2 env)
  | EvalExpr (Constraint(e,t)) env =
        EvalExpr e env
  | EvalExpr (Conditional(e1,e2,e3)) env =
        ApplyIf (EvalExpr e1 env) e2 e3 env
  | EvalExpr (Let(v,e)) env =
        EvalExpr e ((EvalVB env v) @ env)
  | EvalExpr (Select(e,id)) env =
        ApplySelect (EvalExpr e env) id
  | EvalExpr (Class(id,EmptyInheritance,EmptyMethod)) env =
        raise syntax_error
  | EvalExpr (Class(id,EmptyInheritance,Meth(v))) env =
        Fun(fn x => Record(EvalMB ((id,x)::env) v))
  | EvalExpr (Class(id,Inh(h),EmptyMethod)) env =
        Fun(fn x => Record(EvalIN ((id,x)::env) h))
  | EvalExpr (Class(id,Inh(h),Meth(v))) env =
        Fun(fn x => Record((EvalMB ((id,x)::env) v)@(EvalIN ((id,x)::env) h)))

and EvalMB env (ValAnd(h::t)) =
        (EvalMB env (ValAnd(t)))@(EvalMB env h)
  | EvalMB env (ValAnd(nil)) =
        nil
  | EvalMB env (ValEqn(id,e)) =
        [(id,fn v => EvalExpr e ((self,v)::env))]
  | EvalMB env (ValRec(id1,id2,e)) = let
        fun g v = let
```

```
            fun f x = EvalExpr e ([(id2,x),(id1,Fun(f)),(self,v)]@env)
        in
            Fun(f)
        end
    in
        [(id1,g)]
    end

and EvalIN env (InhAnd(h::t)) =
        (EvalIN env (InhAnd(t)))@(EvalIN env h)
  | EvalIN env (InhAnd(nil)) =
        nil
  | EvalIN env (From(id1,id2,e)) =
        InheritMethod (Apply (LookUp(id2,env)) (EvalExpr e env)) id1

and Apply (Fun(f)) v = f v
  | Apply (_) v = raise function_mismatch

and ApplyIf (Constant("true")) et ef env = EvalExpr et env
  | ApplyIf (Constant("false")) et ef env = EvalExpr ef env
  | ApplyIf (_) et ef env = raise condition_mismatch
  ;

fun EvalDecl env (Val(v)) =
        (EvalVB env v) @ env
  | EvalDecl env (Sequence(nil)) =
        env
  | EvalDecl env (Sequence(h::t)) =
        EvalDecl (EvalDecl env h) (Sequence t)
  ;

exception duplicate_method;
exception sort_error;
exception type_error;
exception unify_error;
exception occurs_error;

fun scd f (x,y) = (x,f y);

fun member x nil = false
  | member x (h::t) = if x=h then true else member x t
  ;

fun setdiff nil B = nil
  | setdiff (h::t) B = if member h B then setdiff t B else h::(setdiff t B)
  ;

val EmptySub = (nil,nil);

fun FreeTypeVars(TypeVariable(v)) = [v]
  | FreeTypeVars(Int) = nil
  | FreeTypeVars(Bool) = nil
  | FreeTypeVars(Arrow(t1,t2)) = (FreeTypeVars t1)@(FreeTypeVars t2)
  | FreeTypeVars(Object(p)) = FreeTypeInRows p

and FreeTypeInRows(RowVariable(v)) = nil
  | FreeTypeInRows(Empty) = nil
  | FreeTypeInRows(Ext(id,Absent,p)) = FreeTypeInRows p
  | FreeTypeInRows(Ext(id,Present(t),p)) = (FreeTypeVars t)@(FreeTypeInRows p)
  ;

fun FreeRowInTypes(TypeVariable(v)) = nil
  | FreeRowInTypes(Int) = nil
  | FreeRowInTypes(Bool) = nil
  | FreeRowInTypes(Arrow(t1,t2)) = (FreeRowInTypes t1)@(FreeRowInTypes t2)
  | FreeRowInTypes(Object(p)) = FreeRowVars p

and FreeRowVars(Empty) = nil
  | FreeRowVars(RowVariable(v)) = [v]
  | FreeRowVars(Ext(id,Absent,p)) = FreeRowVars p
  | FreeRowVars(Ext(id,Present(t),p)) = (FreeRowInTypes t)@(FreeRowVars p)
  ;

fun NotFreeTypeIn(x,t) = not(member x (FreeTypeVars(t)));

fun NotFreeRowIn(x,p) = not(member x (FreeRowVars(p)));

local val TCtr = ref ~1 in
  fun NewTypeVar() = (TCtr := !TCtr+1; TypeVariable("'"^(makestring(!TCtr))))
end;

local val ECtr = ref 0 in
  fun NewRowVar() = (ECtr := !ECtr+1; RowVariable("&"^(makestring(!ECtr))))
end;

fun Sub (S as (St,Sp)) (Int) = Int
  | Sub (S as (St,Sp)) (Bool) = Bool
  | Sub (S as (St,Sp)) (Arrow(t1,t2)) = Arrow(Sub S t1,Sub S t2)
  | Sub (S as (St,Sp)) (Object(p)) = Object(SubExt S p)
  | Sub (S as (St,Sp)) (t as TypeVariable(v)) =
        if null St then TypeVariable(v)
        else let val (v',t') = hd St in
            if v = v' then Sub (tl St,Sp) t'
            else Sub (tl St,Sp) t
        end

and SubExt (S as (St,Sp)) (Empty) = Empty
  | SubExt (S as (St,Sp)) (p as RowVariable(v)) =
        if null Sp then RowVariable(v)
        else let val (v',p') = hd Sp in
            if v = v' then SubExt (St,tl Sp) p'
            else SubExt (St,tl Sp) p
        end
  | SubExt (S as (St,Sp)) (Ext(id',Absent,p')) =
        Ext(id',Absent,SubExt S p')
  | SubExt (S as (St,Sp)) (Ext(id',Present(t'),p')) =
        Ext(id',Present(Sub S t'),SubExt S p')
  ;

fun Unify (S as (St,Sp)) (t1,t2) = let
    val t1' = Sub S t1 and t2' = Sub S t2
in
    case (t1',t2') of
        (Int,Int) => S
      | (Bool,Bool) => S
      | (Arrow(x,y),Arrow(x',y')) => Unify (Unify S (y,y')) (x,x')
      | (Object(p),Object(p')) => UnifyExt S (p,p')
      | (TypeVariable v,TypeVariable u) =>
            if v=u then S
            else (St@[(v,t2')],Sp)
      | (_,TypeVariable v) =>
            if NotFreeTypeIn(v,t1') then (St@[(v,t1')],Sp)
            else raise occurs_error
      | (TypeVariable v,_) =>
            if NotFreeTypeIn(v,t2') then (St@[(v,t2')],Sp)
            else raise occurs_error
      | (_,_) => raise unify_error
end
```

```
and UnifyFld S (Absent,Absent) = S
  | UnifyFld S (Present(t1),Present(t2)) = Unify S (t1,t2)
  | UnifyFld S (_,_) = raise unify_error

and UnifyExt (S as (St,Sp)) (p1,p2) = let
    fun Sort (Empty) = (Empty,Empty)
      | Sort (RowVariable(v)) = (RowVariable(v),RowVariable(v))
      | Sort (Ext(id,f,p)) = let
            val (p',e) = (Sort p);
            fun Insert id f (p as Empty) = Ext(id,f,Empty)
              | Insert id f (p as RowVariable(v)) = Ext(id,f,RowVariable(v))
              | Insert id f (p as Ext(id',f',p')) =
                    if id < id' then Ext(id,f,Insert id' f' p')
                    else if id > id' then Ext(id',f',Insert id f p')
                    else raise duplicate_method
        in
            (Insert id f p',e)
        end
    ;
    val (p1',e1) = Sort(SubExt S p1) and (p2',e2) = Sort(SubExt S p2);
    fun Pad S (p1 as Ext(id,f,p)) p2 (Empty) =
            UnifyExt (UnifyFld S (f,Absent)) (p,p2)
      | Pad S (p1 as Ext(id,f,p)) p2 (e as RowVariable(v)) =
            UnifyExt (UnifyExt S (Ext(id,f,NewRowVar()),e)) (p1,p2)
      | Pad S _ p2 _ =
            raise sort_error
in
    case (p1',p2') of
        (Empty,Empty) => S
      | (Ext(id1,f1,p1''),Ext(id2,f2,p2'')) =>
            if id1=id2 then UnifyExt (UnifyFld S (f1,f2)) (p1'',p2'')
            else if id1<id2 then Pad S p1' p2' e2
            else Pad S p2' p1' e1
      | (RowVariable v,RowVariable u) =>
            if v=u then S
            else (St,Sp@[(v,RowVariable u)])
      | (_,RowVariable v) =>
            if NotFreeRowIn(v,p1') then (St,Sp@[(v,p1')])
            else raise occurs_error
      | (RowVariable v,_) =>
            if NotFreeRowIn(v,p2') then (St,Sp@[(v,p2')])
            else raise occurs_error
      | (_,_) => raise unify_error
end;

fun genUpdate (A as (AS,NG)) nil = A
  | genUpdate (A as (AS,NG)) ((id,t)::T) = genUpdate ((id,t)::AS,NG) T
  ;

fun nongenUpdate (A as (AS,NG)) nil = A
  | nongenUpdate (A as (AS,NG)) ((id,t)::T) = nongenUpdate ((id,t)::AS,t::NG) T
  ;

fun Fresh NG t = let
        fun NewTypes id = (id,NewTypeVar());
        fun NewRows id = (id,NewRowVar());
        fun f (H::T) g = (g H)@(f T g)
          | f nil g = nil
          ;
        val BVtype = f NG FreeTypeVars;
        val BVext = f NG FreeRowInTypes;
        val FVtype = setdiff (FreeTypeVars t) BVtype;
        val FVext = setdiff (FreeRowInTypes t) BVext;
        val S = (map NewTypes FVtype,map NewRows FVext)
    in
        (Sub S t)
    end;

fun TypeVB (A as (AS,NG)) S (ValEqn(id,e)) = let
        val (S',t') = TypeExpr A S e
    in
        (S',genUpdate A [(id,t')])
    end
  | TypeVB (A as (AS,NG)) S (ValRec(id1,id2,e)) = let
        val alpha = NewTypeVar() and beta = NewTypeVar();
        val A1 = nongenUpdate A [(id1,Arrow(beta,alpha)),(id2,beta)];
        val (S1,t1) = TypeExpr A1 S e;
        val S2 = Unify S1 (alpha,t1)
    in
        (S2,genUpdate A [(id1,Sub S2 (Arrow(beta,t1)))])
    end
  | TypeVB (A as (AS,NG)) S (ValAnd(h::t)) = let
        val (S1,(AS1,NG1)) = TypeVB A S h;
        val (S2,(AS2,NG2)) = TypeVB A S1 (ValAnd(t))
    in
        (S2,(AS2@AS1,NG2@NG1))
    end
  | TypeVB (A as (AS,NG)) S (ValAnd(nil)) = (S,A)

and TypeExpr (A as (AS,NG)) S (Identifier(id)) = (S,Sub S (Fresh NG (LookUp(id,AS))))
  | TypeExpr (A as (AS,NG)) S (Self) = (S,Sub S (Fresh NG (LookUp(self,AS))))
  | TypeExpr (A as (AS,NG)) S (Abstraction(id,e)) = let
        val alpha = NewTypeVar();
        val A1 = nongenUpdate A [(id,alpha)];
        val (S',t') = TypeExpr A1 S e
    in
        (S',Arrow(Sub S' alpha,t'))
    end
  | TypeExpr (A as (AS,NG)) S (Application(e1,e2)) = let
        val (S1,t1) = TypeExpr A S e1;
        val (S2,t2) = TypeExpr A S1 e2;
        val alpha = NewTypeVar();
        val S3 = Unify S2 ((Sub S2 t1),(Arrow(t2,alpha)))
    in
        (S3,Sub S3 alpha)
    end
  | TypeExpr (A as (AS,NG)) S (Constraint(e,t)) = let
        val (S1,t1) = TypeExpr A S e;
        val S2 = Unify S1 (t1,(Sub S1 t))
    in
        if Sub S2 t = t then (S2,t)
        else raise type_error
    end
  | TypeExpr (A as (AS,NG)) S (Let(v,e)) = let
        val (S1,(AS1,NG1)) = TypeVB A S v;
        val A2 = (map (scd (Sub S1)) AS1,map (Sub S1) NG1);
        val (S2,t2) = TypeExpr A2 S1 e
    in
        (S2,t2)
    end
  | TypeExpr (A as (AS,NG)) S (Conditional(e1,e2,e3)) = let
        val (S1,t1) = TypeExpr A S e1;
        val S1' = Unify S1 (Bool,t1);
        val (S2,t2) = TypeExpr A S1' e2;
        val (S3,t3) = TypeExpr A S2 e3;
        val S4 = Unify S3 (t2,t3)
    in
```

```
        (S4,Sub S4 t3)
    end
  | TypeExpr (A as (AS,NG)) S (Select(e,id)) = let
        val alpha = NewTypeVar();
        val rho = NewRowVar();
        val (S1,t1) = TypeExpr A S e;
        val S2 = Unify S1 (t1,Object(Ext(id,Present(alpha),rho)))
    in
        (S2,Sub S2 alpha)
    end
  | TypeExpr (A as (AS,NG)) S (Class(id,EmptyInheritance,EmptyMethod)) =
        (EmptySub,Arrow(NewTypeVar(),Object(Empty)))
  | TypeExpr (A as (AS,NG)) S (Class(id,EmptyInheritance,Meth(m))) = let
        val alpha = NewTypeVar() and beta = NewTypeVar();
        val A1 = nongenUpdate A [(id,alpha),(self,beta)];
        val (S1,p) = TypeMB A1 S m Empty;
        val S2 = Unify S1 (beta,Object(p))
    in
        (S2,Sub S2 (Arrow(alpha,beta)))
    end
  | TypeExpr (A as (AS,NG)) S (Class(id,Inh(i),EmptyMethod)) = let
        val alpha = NewTypeVar() and beta = NewTypeVar();
        val A1 = nongenUpdate A [(id,alpha),(self,beta)];
        val (S1,p) = TypeIN A1 S i Empty;
        val S2 = Unify S1 (beta,Object(p))
    in
        (S2,Sub S2 (Arrow(alpha,beta)))
    end
  | TypeExpr (A as (AS,NG)) S (Class(id,Inh(i),Meth(m))) = let
        val alpha = NewTypeVar() and beta = NewTypeVar();
        val (A1 as (AS1,NG1)) = nongenUpdate A [(id,alpha),(self,beta)];
        val (S1,p1) = TypeIN A1 S i Empty;
        val A2 = (map (scd (Sub S1)) AS1,map (Sub S1) NG1);
        val (S2,p2) = TypeMB A2 S1 m p1;
        val S3 = Unify S2 (beta,Object(p2))
    in
        (S3,Sub S3 (Arrow(alpha,beta)))
    end

and TypeMB (A as (AS,NG)) S (ValEqn(id,e)) p = let
        val (S',t') = TypeExpr A S e
    in
        (S',Ext(id,Present(t'),p))
    end
  | TypeMB (A as (AS,NG)) S (ValRec(id1,id2,e)) p = let
        val alpha = NewTypeVar() and beta = NewTypeVar();
        val A1 = nongenUpdate A [(id1,Arrow(beta,alpha)),(id2,beta)];
        val (S1,t1) = TypeExpr A1 S e;
        val S2 = Unify S1 (alpha,t1)
    in
        (S2,Ext(id1,Present(Sub S2 (Arrow(beta,t1))),p))
    end
  | TypeMB (A as (AS,NG)) S (ValAnd(h::t)) p = let
        val (S1,p1) = TypeMB A S h p
    in
        TypeMB A S1 (ValAnd(t)) p1
    end
  | TypeMB (A as (AS,NG)) S (ValAnd(nil)) p = (S,p)

and TypeIN (A as (AS,NG)) S (From(id1,id2,e)) p = let
        val (S1,t1) = TypeExpr A S e;
        val t2 = Sub S1 (Fresh NG (LookUp(id2,AS)));
        val gamma = NewTypeVar();
        val rho = NewRowVar();
        val S2 = Unify S1 (t2,Arrow(t1,Object(Ext(id1,Present(gamma),rho))))
    in
        (S2,Ext(id1,Present(Sub S2 gamma),p))
    end
  | TypeIN (A as (AS,NG)) S (InhAnd(h::t)) p = let
        val (S1,p1) = TypeIN A S h p
    in
        TypeIN A S1 (InhAnd(t)) p1
    end
  | TypeIN (A as (AS,NG)) S (InhAnd(nil)) p = (S,p)
  ;

fun TypeDecl (A as (AS,NG)) (Val(v)) = let
        val (S1,(AS1,NG1)) = (TypeVB A EmptySub v)
    in
        (map (scd (Sub S1)) AS1,map (Sub S1) NG1)
    end
  | TypeDecl (A as (AS,NG)) (Sequence(nil)) =
        A
  | TypeDecl (A as (AS,NG)) (Sequence(h::t)) =
        TypeDecl (TypeDecl A h) (Sequence(t))
  ;
```