Documentation for `polyrec_sml`: an extension SML with typechecking for polymorphic recursion

Martin Emms
Centrum für Informations- und Sprachverarbeitung
Universität München, Wagmüllerstr 23, D-80538 München
email: emms@cis.uni-muenchen.de

July, 1995

Summary: The usual rule used for typechecking recursive declarations can be called *monomorphic*, because in typing the declaration, the recursively bound variable is assumed to have monomorphic type. This forces all occurrences of the variable to have the same type and prevents certain recursions from being typed. `polyrec_sml` is a version of the SMLofNJ compiler which implements a more powerful, *polymorphic* recursion typing rule, first proposed in [Mycroft, 1984]. This essentially allows different occurrences of the recursively bound variable to take on different instances of the type of the definition. The implementation allows the user to choose between monomorphic and polymorphic recursion, with an attendant $\sim$ 10 % effect on total compile time. It is hoped the implementation will enable further experience to be gathered on programming with polymorphic recursion.

Sources, installation notes and examples for `polyrec_sml` can be obtained from:

`ftp.cis.uni-muenchen.de/incoming/emms/polyrec_dist`

Remarks concerning `polyrec_sml` can be assured of an interested reception at:

`emms@cis.uni-muenchen.de`
`leiss@cis.uni-muenchen.de`

1 2

[1] Thanks to Hans Leiß for his helpful comments on this documentation. The fault for remaining errors remains with the author. The implementation described here was created with the support of DFG project 'Semiunifikation' Le 788/1-2.

[2] *Added October 1996*: this report touches only lightly on theoretical aspects and primarily documents the `polyrec_sml` implementation. In subsequently trying to prove its correctness, some changes to the basic algorithm were found to be necessary. Correctness of the variant algorithm is proved in [Emms and Leiss, forthcoming] and it is hoped a later version of `polyrec_sml` will incorporate the changes. The changes have an effect only when (i) two recursions are nested and (ii) the inner recursion is of the kind that gets a more general type under polymorphic recursion than under monomorphic.

Contents

The code for `polyrec_sml` is adapted from, or is intended to be used in combination with, the code which consitututes Standard ML of New Jersey, Version 0.93. It is therefore issued with the following copyright notice, license and disclaimer.

Standard ML of New Jersey Copyright Notice, License and Disclaimer.

Installation notes for `polyrec_sml` are in *doc/INSTALL_NOTES*, and a brief user-manual in *doc/polyrec_man*. The current document gives some theoretical background, and documents in detail the implementation.

Section 1 defines and illustrates what type-checking wrt. polymorphic recursion is, confining attention to a sublanguage of SML. Section 2 defines a Damas-Milner style type-checker for polymorphic recursion, incorporating semiunification. This theoretical version of the type-checker is at the same distance from our implementation as the standard Damas-Milner algorithm is from the standard SMLofNJ implementation. Sections 3 and 4 are directly concerned with the implementation. In section 3 we describe a prototype of the type-checker in an endeavour to keep the outline visible amidst the detail. Section 4 documents the code, both the additional code, and some of the relevant pre-existing code. Section 5 contains some concluding remarks on the design, and section 6 desribes the performance.

1 Polymorphic Recursion

In this section we define and give examples of, typing wrt. polymorphic recursion.

1.1 The Typing Calculi ML and ML^+

To keep the description of polymorphic recursion manageable, we will confine attention to a sublanguage of SML which exhibits the relevant features. In defining the language we will use metavariables as follows:

x	: variables
c	: constructors (arity 0 $\equiv$ constants, binary infix `(_,_)` and `(_;_)` presupposed)
e	: expressions
d	: declarations
α	: type-variables
κ	: type-function (arity 0 $\equiv$ type constant, '*list*' presupposed unary, '$\rightarrow$' and '*' presupposed infix binary)
tn	: type-function name
τ	: quantifier-free type
$\overline{\tau}$	: quantified type

The types and (untyped) expressions and declarations of ML are:

$$\begin{array}{lcl}
\tau & := & \alpha \mid \vec{\tau}\kappa \\
\overline{\tau} & := & \tau \mid \forall\alpha.\overline{\tau} \\
e & := & x \mid c \mid ee' \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid \texttt{let}\ d\ \texttt{in}\ e \\
d & := & \texttt{val}\ x\ \texttt{=}\ e \\
 & \mid & \texttt{val rec}\ x\ \texttt{=}\ e \\
 & \mid & \texttt{datatype}\ \vec{\alpha}\ tn\ \texttt{=}\ c_1 \mid \ldots c_m \mid c_{m+1}\ \texttt{of}\ \tau_{m+1} \mid \ldots c_{m+n}\ \texttt{of}\ \tau_{m+n}
\end{array}$$

A *type-assignment calculus* operates as a further filter on ML code, defining a set of possible types for a piece of code, relative to a type-context. The first six typing rules in Figure 1.1 define the calculus for ML (which we will also refer to as ML).

Notation conventions. E is a type-context, assigning types to constructors and variables (and type functions to type names). $\overline{\tau} \succ \tau$ (read τ 'instantiates' $\overline{\tau}$) holds if τ results from $\overline{\tau}$ by instantiating all bound variables of $\overline{\tau}$ with monomorphic ($\forall$-free) types. $\overline{\tau}^E$ means quantifying all variables of τ except those occurring free in type assumptions in E. By $\overline{E}^{E'}$ we mean replacing each $x : \tau$ in E with $x{:}\overline{\tau}^{E'}$. In the `let` rule, $E + E'$ is understood as the destructive overwriting of E by E'. The context will be assumed to type the pairing and sequencing constructors as $\forall\alpha\forall\beta.\alpha{\rightarrow}\beta{\rightarrow}(\alpha * \beta)$ and $\forall\alpha\forall\beta.\alpha{\rightarrow}\beta{\rightarrow}\beta$.

Special attention should be paid to the rule for recursive value declarations, in which the recursively bound variable is assumed in the premises to have a *monomorphic* type. Although the declaration may well *assign* a polymorphic type to the recursive function, this polymorphic type may not be *assumed* during the typing of the declaration. The final rule above is the alternative so-called 'Polymorphic Recursion' typing rule, first considered in [Mycroft, 1984], which allows that the assumed type for the recursively bound variable can be polymorphic.

The typing calculus with this rule we will refer to as ML^+, and we illustrate the difference between ML and ML^+ by means of some examples in the following section. Before that we define the notion of principal type and environment.

Definition 1 (Principal Types and Environments) *Let $\mathcal{K}$ be ML or ML^+, e an expression and d a declaration.*

i) τ is the $\mathcal{K}$-principal type of e wrt to E if $\mathcal{K}$ proves $E \mid\!\!- e : \tau$, and for any τ' such that $\mathcal{K}$ proves $E \mid\!\!- e : \tau'$, $\tau' = T\tau$, for some substitution T of monomorphic types for type variables.

ii) $\{x \mapsto \forall\vec{\alpha}\tau\}$ is the $\mathcal{K}$-principal environment for d wrt to E if $\mathcal{K}$ proves $E \mid\!\!- d : \{x \mapsto \forall\vec{\alpha}\tau\}$, and for any τ' such that $\mathcal{K}$ proves $E \mid\!\!- d : \{x \mapsto \tau'\}$, $\tau' = \overline{T\tau}^E$, for some substitution T.

For example, with respect to the empty environment $\{\}$, the ML-principal environment for the declaration `val f = fn x => x` is $\{f \mapsto \forall\alpha.\alpha{\rightarrow}\alpha\}$. Another environment generated by the declaration is $\{f \mapsto \forall\alpha.(int * \alpha){\rightarrow}(int * \alpha)\}$, and we have $\overline{T(\alpha{\rightarrow}\alpha)}^{\{\}} = \forall\alpha.(int * \alpha){\rightarrow}(int * \alpha)$ for $T(\alpha) = (int * \alpha)$.

1.2 Examples

In this section a series of examples are given where the ML-principal environment and the ML^+ principal environment differ. The source code for these and other examples is in *doc/examples*, and *doc/compiler_examples* lists some further examples that arise in the source code for the SML compiler.

Example 1

```
val rec f = fn x => f 1
```

Although this is is an artifial example, the defined `f` being non-terminating, it makes a good starting point for illustrating the typing rules. It is clear that (i) in order to type the definition, E must contain an assumption for `f` which has $int{\rightarrow}\beta$ as a possible instantiation, and that (ii) the possible types of the definition, assuming such an instantiation, are the instances of $\alpha{\rightarrow}\beta$. With ML comes then the additional constraint that the assumed type be IDENTICAL TO THE DEFINITION TYPE (and therefore monomorphic). Thus the assumption for `f` must be an instance of $int{\rightarrow}\beta$, and in the ML-principal environment, `f` will be assigned $\forall\beta.int{\rightarrow}\beta$.

Standard ML typing rules

Choose
$$\frac{E(x) \succ \tau}{E \mid\!- x : \tau}, \quad x \text{ is either a constant or variable}$$

App
$$\frac{E \mid\!- \mathtt{e}_1 : \tau' \to \tau \quad E \mid\!- \mathtt{e}_2 : \tau'}{E \mid\!- \mathtt{e}_1 \mathtt{e}_2 : \tau}$$

Functions
$$\frac{E + \{x \mapsto \tau'\} \mid\!- \mathtt{e} : \tau}{E \mid\!- \mathtt{fn\ x\ =>\ e} : \tau' \to \tau}$$

Let
$$\frac{E \mid\!- d : E' \quad E + E' \mid\!- e : \tau}{E \mid\!- (\mathtt{let}\ d\ \mathtt{in}\ e\ \mathtt{end}) : \tau}$$

Datatype Dec
$$\frac{\begin{array}{l} TE = \{\mathrm{tn} \mapsto \kappa\} \\ E_1 = \{\mathtt{c}_1 \mapsto \forall\vec{\alpha}.\vec{\alpha}\kappa, \ldots, \mathtt{c}_n \mapsto \forall\vec{\alpha}.\vec{\alpha}\kappa\} \\ E_2 = \{\mathtt{c}_{m+1} \mapsto \forall\vec{\alpha}.\tau_{m+1}[\kappa/tn] \to \vec{\alpha}\kappa, \ldots, \mathtt{c}_{m+n} \mapsto \forall\vec{\alpha}.\tau_{m+n}[\kappa/tn] \to \vec{\alpha}\kappa\} \end{array}}{\begin{array}{l} E \mid\!- \mathtt{datatype}\ \vec{\alpha}\ tn = c_1 \mid \ldots c_m \mid c_{m+1}\ \mathtt{of}\ \tau_{m+1} \mid \ldots c_{m+n}\ \mathtt{of}\ \tau_{m+n} \\ \qquad : TE + E_1 + E_2 \end{array}}$$

Value Dec
$$\frac{E \mid\!- e : \tau}{E \mid\!- \mathtt{val}\ x\ \mathtt{=}\ e : \{x \mapsto \overline{\tau}^E\}}$$

Rec Value Dec
$$\frac{E + \{f \mapsto \tau\} \mid\!- e : \tau}{E \mid\!- \mathtt{val\ rec}\ f\ \mathtt{=}\ e : \{f \mapsto \overline{\tau}^E\}}$$

Alternative ML^+ rule

Rec Value Dec (Poly)
$$\frac{E + \{\mathtt{x} \mapsto \overline{\tau}^E\} \mid\!- \mathtt{e} : \tau}{E \mid\!- \mathtt{val\ rec\ x\ =\ e} : \{\mathtt{x} \mapsto \overline{\tau}^E\}}$$

Figure 1: The typing calculi ML and ML^+

On the other hand, working with ML^+, we have the constraint that the assumed type be THE CLOSURE OF THE DEFINITION TYPE. Clearly $\forall\alpha\forall\beta.\alpha\rightarrow\beta$ fulfils the requirements, and will be the type assigned to `f` in the ML^+ principal environment[3]. If the example is slightly changed to

```
val rec f = fn x => (f 1 ; f "a")
```

the difference is more stark: the code is ML-*un*typable, whilst the ML^+-principal environment remains the same.

Essentially this situation can realistically arise in a simultaneous recursive definition, as noted in [Mycroft, 1984]:

Example 2

```
fun map f l = if l = [] then [] else (f(hd(l))::(map f (tl l)))
and incrlist l = map (fn x => 1 + x) l
```

Inside `incrlist`'s definition, `map` requires type $(int\rightarrow int)\rightarrow int\ list\rightarrow int\ list$. This is a special case of the type $(\alpha\rightarrow\beta)\rightarrow\alpha\ list\rightarrow\beta\ list$ that is required for `map`'s occurrence inside its own definition. Under ML, the more specific type must be assumed for `map` in any typing derivation, and in the ML-principal environment, `map` will therefore be assigned $(int\rightarrow int)\rightarrow int\ list\rightarrow int\ list$. However, using the usual polymorphic type for `map` as the assumption, both occurrences of `map` can be typed, and the assumption is indeed the closure of the type of its definition. In the ML^+-principal environment, `map` therefore receives the usual type, $\forall\alpha\forall\beta.(\alpha\rightarrow\beta)\rightarrow\alpha\ list\rightarrow\beta\ list$.

Again, if the example is slightly modified by adding a further clause

```
and prefix l = map (fn x => "a"^x) l
```

with the aim of defining three functions simultaneously, the difference is starker: `map` then occurs with two incompatible types in the definitions of the two other functions, and so the whole declaration is ML-*un*typable. The ML^+-principal environment remains the same, because the type of the additional occurrence is just another instance of the closure of `map`'s definition type.

The above case was an example of a non-essentially simultaneous definition, and if the functions are separately defined, they get the same types under ML and ML^+. In *doc/examples*, the difference between ML and ML^+ is further illustrated by genuinely mutually recursive definitions, some of which occurred in programming practice[4] and are only typable with respect to ML^+. Also, *compiler_examples* contains many examples of ML-typable simultaneous recursions, drawn from the SML compiler code, which have more general ML^+-types.

Another class of examples arises when datatypes $\alpha\ \kappa$ are used where a subcomponent of a $\alpha\ \kappa$ value can be a $\alpha\ \kappa\ \kappa$ value. The first example of this below derives from a message [Elliot, 1991] to the SML email list (see also *doc/examples*)

Example 3

```
datatype entry = Const of const | N of (entry * entry)
datatype 'a c_dtree = EC | C of (const * 'a * 'a c_dtree)
datatype 'a e_dtree = ET | D of ('a c_dtree * ('a e_dtree) e_dtree)
```

[3] In *doc/examples* there are cases from the compiler sources of this kind, where an occurrence of the recursively bound variable in some side-effecting part of the definition restricts the type more than occurrences in the definition proper.

[4] one of these was reported to the ML-list as recently as July 1995 [Altenkirch, 1995].

```
fun ed_find (D (C(con',v,rest),_),Const(con)) =
    if con = con' then v
    else ed_find (D(rest, ET), Const(con))
  | ed_find (D(_,dtree), N(e,erest)) =
    ed_find (ed_find (dtree,e),erest)
```

The purpose of the code is to assign values to tuples (represented using the `entry` datatype). When there is overlap amongst the tuples, simply writing a rule with a case for every tuple will be inefficient, because while stepping through the cases, certain parts of the input may be repeatedly matched against the recurring parts of the case definitions. An alternative is to uses a *discrimination tree* of type α *e_dtree*, having two parts. The first is an α *c_tree*, which associates values to the constants which appear in an input tuple. The second part is of type (α *e_dtree*) *e_dtree*, and the significant component will be an (α *e_dtree*) *c_dtree*, assigning a α *e_dtree* to constants. `ed_find` traverses a tuple, each time using the first element to select a particular α *d_tree* as the discrimination tree for the remainder. In this way a call of `ed_find` on an argument of type α *e_dtree e_dtree* supplies an argument of type α *e_dtree* to a further call of `ed_find`. This makes `ed_find` ML-*un*typable, whilst it has type $\forall\alpha.\alpha$ *e_dtree* $*$ *entry*$\rightarrow\alpha$ wrt ML^+.

Example 4 Wadsworth more than 10 years ago is reported to have encountered the analogous problem in a 'real' program, making use of the datatype `'a T = Empty | Node of 'a * ('a T) T`. Suppose the environment supplies `fst`:$\forall\alpha.\alpha T\rightarrow\alpha$ and `snd`:$\forall\alpha.\alpha\ T\rightarrow\alpha\ T\ T$, `flatten`: α *list list*$\rightarrow\alpha$ *list*.

```
collect x =   if x = Empty then []
  else fst x ::  (flatten (map collect (collect (snd x))))
```

It is a useful exercise to consider in detail the typing of this example. Clearly `x` must have type $\alpha\ T$. For a ML-typing, we obtain after some calculation that a monomorphic type τ must be assumed for `collect`, satisfying the instantiation claims:

$$\tau \succ \alpha\ T\ T\rightarrow\beta\ list$$
$$\tau \succ \beta\rightarrow\alpha\ list$$

Since τ is monomorphic, $\succ$ can be read as identity, and the righthand sides equated. This requires the impossible identity $\alpha\ T\ T\ list = \alpha\ list$, and so the example is ML-*un*typable.

For a ML^+ typing, we must assume a polymorphic $\overline{\tau}$ satisfying the instantiation claims $\overline{\tau} \succ \alpha\ T\ T\rightarrow\beta\ list$ and $\overline{\tau} \succ \beta\rightarrow\alpha\ list$. Additionally, $\overline{\tau}$ must be the closure of some type of the definition. Assuming the instantiation claims are satisfied, the possible types of the definition are the instances of $\alpha\ T\rightarrow\alpha\ list$. Hence for a typing we require a substitution S such that

$$\overline{S(\alpha\ T\rightarrow\alpha\ list)}^{SE} \succ S(\alpha\ T\ T\rightarrow\beta\ list)$$
$$\overline{S(\alpha\ T\rightarrow\alpha\ list)}^{SE} \succ S(\beta\rightarrow\alpha\ list)$$

The substitution $S = [\alpha\ T/\beta]$, is a solution, in fact a most general. Therefore in the ML^+-principal environment, `collect` is assigned the type, $\overline{S(\alpha\ T\rightarrow\alpha\ list)}^{SE} = \overline{\alpha\ T\rightarrow\alpha\ list}^{E} = \forall\alpha.\alpha\ T\rightarrow\alpha\ list$.

We consider one more example below to illustrate the interaction between λ-binding and polymorphic recursion.

Example 5

```
fn y => let fun collect x = (x = y;fst x ::  (flatten (map collect (collect
(snd x))))) in ...
```

Note that now the argument of `collect` must have the same type as the wider-scoped, abstracted variable `y`. Arguing exactly as above, in order to type the embedded declaration, we will come to the requirement that there be a S such that:

$$\overline{S(\alpha\ T{\rightarrow}\alpha\ list)}^{S(E+y:\alpha)} \succ S(\alpha\ T\ T{\rightarrow}\beta\ list)$$
$$\overline{S(\alpha\ T{\rightarrow}\alpha list)}^{S(E+y:\alpha)} \succ S(\beta{\rightarrow}\alpha\ list)$$

The presence of α in a type assumption means that the closure here is vacuous, and the problem is equivalent to the unsolvable:

$$S(\alpha\ T{\rightarrow}\alpha\ list) \succ S(\alpha\ T\ T{\rightarrow}\beta\ list)$$
$$S(\alpha\ T{\rightarrow}\alpha\ list) \succ S(\beta{\rightarrow}\alpha\ list)$$

As already noted, the source code for these and further examples is given in *doc/examples*, whilst *doc/compiler_examples* notes some of the functions from the compiler code that receive more general types under ML^+ than ML.

2 Theory of Polymorphic Recursion

For ML and ML^+ we have the following:

Theorem 1 (Existence of Principal Types and Environments) *Where $\mathcal{K}$ is ML or ML^+ for any context E, and any expression (resp. declaration) t, if t is $\mathcal{K}$-typable wrt E, then t has a $\mathcal{K}$-principal type (resp. environment).*

Theorem 2 (ML^+ generalises ML) *If t is an ML-typable expression (resp. declaration), then t is ML^+ typable, and the ML^+-principal type (resp. environment) is more general than the ML-principal type (resp. environment)*

The ML case of Theorem 1 is proved in [Damas and Milner, 1982], by means of a recursive type-assignment function, $\mathcal{W}$, essentially first defined in [Milner, 1978], which takes an environment and code and returns either FAIL or the least environment revision required for ML-typability, and the corresponding ML-principal type or environment. The FAIL value implies *ML-untypability* under any revision of the environment.

The ML^+ case of of Theorem 1 is proved in [Mycroft, 1984], by means of an adaption of $\mathcal{W}$ to a (semi-) algorithm, in which the unknown polymorphic assumptions in recursive declarations are obtained by an iterative procedure which in the case of typable functions reaches a fixed-point, but which diverges on all untypable functions[5].

An alternative semi-algorithm for ML^+ will be described below, based on the work of [Henglein, 1988] and [Leiss, 1989], which invokes a procedure to solve a so-called *semiunification* problem. Recall in the `collect` example above, the typing problem reduced to the existence of a substitution, S, solving instantiation claims of the form $\overline{S\tau}^{SE} \succ S\sigma$. Such a problem is essentially a *semiunification* problem, and the solution a *semiunifier*. In this section we give a formal definition of semiunificaton and of a type assignment function for ML^+.

[5] Further conditions were proposed to avoid loopings, but these caused also some typable code to be rejected.

2.1 The idea

As the examples will perhaps have made clear, there will be essentially two phases in typechecking a recursive declaration `val rec f = (... f1 ... fn ...)`:

Phase 1: the definition part, `(... f1 ... fn ...)` is typed effectively assuming the recursively bound variable, `f`, has type $\forall\alpha.\alpha$ (this was also the first step in Mycroft's iterative procedure). Thus one ignores to begin with the fact that the polymorphic assumption for `f` has to be the closure of some type of the definition. In order to be able to attend to this in Phase 2, a record is made of the instantiations of $\forall\alpha.\alpha$, that are required to type the occurrences of `f` (this is not a part of Mycroft's procedure). After the defining term has been typed we have 3 outputs:

$L = \forall\alpha.\alpha \succ \tau_1, \ldots, \forall\alpha.\alpha \succ \tau_n$,	a record of instantiation claims generated by typing the occurrences $\mathtt{f}_1, \ldots, \mathtt{f}_n$, of the recursively bound variable,
S,	a substitution, of monotypes for variables, by which it may be necessary to refine the *free* type variables in the environment in order to type the definition, and
τ,	the type of the definition `(... f1 ... fn ...)`, under the assumption, `f`:$\forall\alpha.\alpha$.

Phase 2: τ and L contain all the information necessary to now determine the typability of the recursive declaration. Any typing of the defining term, given some polymorphic assumption, clearly must be by means of some specialisation U of the τ_i. Furthermore, ML^+ requires a specialisation, U, with a particular property: when the closure of the resulting type for the defining term (i.e. $\overline{U\tau}^{USE}$) is taken for the polymorphic assumption instead of $\forall\alpha.\alpha$, the (specialised) instantiation claims should be true. A substitution U is therefore required such that:

$$\overline{U\tau}^{USE} \succ U\tau_1, \ldots, \overline{U\tau}^{USE} \succ U\tau_n$$

This problem is essentially a *semiunification* problem (see following section), and has a most general solution if it has any solution at all. Phase 2 thus consists of the conversion of the true set instantiation claims, L, into the specification of a further set of instantiation claims, and then finding the most general solution for the unknown substitution, U. The final output of the type checker on a recursive declaration, in terms of this U will be:

$L' = \overline{U\tau}^{USE} \succ U\tau_1, \ldots, \overline{U\tau}^{USE} \succ U\tau_n$,

US,	the net environment-revising substitution, necessary to type the declaration, and
$\{\mathtt{f} \mapsto \overline{U\tau}^{USE}\}$,	the environment generated by the declaration.

There are certain important further details due to the embedding of declarations in `let` expressions, and the full specification appears in section 2.3. The following section defines the key notion of *semiunification*.

2.2 Semiunification

Definition 2 (Matching) *$\tau \sqsubseteq_C \sigma$ holds between two monomorphic types if there is some substitution T (called the* matching substitution*) which is an identity on the variables in C, taking τ into σ. C will be referred to as the* context *variables. Those free variables α of τ, for which $T\alpha \neq \alpha$, will be called the* pattern *variables.*

Definition 3 (Semiunification Problem) *is a multi-set of equalities, $\tau = \sigma$, and inequalities, $\tau \sqsubseteq_i \sigma$, where $1 \leq i \leq n$, where τ and σ are monomorphic. A solution relative to a context set C is an $n+1$ tuple $(U, T_1, \ldots, T_n)$ of substitutions, such that U specialises the equations to true identities, and each inequation $\tau \sqsubseteq_i \sigma$, to a true match, $U\tau \sqsubseteq_{UC} U\sigma$, T_i being the associated matching substitution. A solution $(U, T_1, \ldots, T_n)$ is said to be more general than $(U', T'_1, \ldots, T'_n)$, if U' is a specialisation of U.*

Where L is a sequence of $\sqsubseteq_C$ match claims, we define $\overline{L}^C = \{\overline{\tau}^E \succ \sigma : \tau \sqsubseteq_C \sigma \in L\}$, where E is any environment the free type variables of whose type assumptions $= FV(C)$. The reason that typing with respect to polymorphic recursion can be reduced to semiunification is:

> $\sigma \sqsubseteq_C \tau$ holds iff $\overline{\sigma}^E \succ \tau$ where E is any environment the free type variables of whose type assumptions $= FV(C)$.

The tool used to find a solution to a semiunification problem is a rewrite system [Henglein, 1988], [Leiss, 1989][6]. In this system, equations and inequations are rewritten till either FAIL is generated or no further rewrites are possible. Whenever no further rewrites are possible, a solving 'semiunifier' for the original system can be read off.

Definition 4 (The rewrite system)

U1. $C, L \wedge (\tau_1 * \ldots * \tau_m)\kappa_1 = (\sigma_1 * \ldots * \sigma_n)\kappa_1$
$\quad \mapsto FAIL$, *if* $\kappa_1 \neq \kappa_2$
$\quad \mapsto C, L \wedge \tau_1 = \sigma_1 \wedge \ldots \wedge \tau_n = \sigma_n$, *otherwise*

U2. $C, L \wedge \vec{\tau}\kappa = \alpha \mapsto C, L \wedge \alpha = \vec{\tau}\kappa$

U3. $C, L \wedge \alpha = \vec{\tau}\kappa \mapsto FAIL$ *if* $\alpha \in FV(\vec{\tau})$

U4. $C, L \wedge \alpha = \alpha \mapsto C, L$

U5. $C, L \wedge \alpha = \tau \mapsto C[\tau/\alpha], L[\tau/\alpha] \wedge \alpha = \tau$, *if* $\alpha \notin FV(\tau)$, $\alpha \in FV(L)$

S1. $C, L \wedge (\tau_1 * \ldots * \tau_m)\kappa_1 \sqsubseteq_i (\sigma_1 * \ldots * \sigma_n)\kappa_2$
$\quad \mapsto FAIL$, *if* $\kappa_1 \neq \kappa_2$
$\quad \mapsto C, L \wedge \tau_1 \sqsubseteq_i \sigma_1 \wedge \ldots \wedge \tau_n \sqsubseteq_i \sigma_n$ *otherwise*

S2, $C, L \wedge \alpha \sqsubseteq_i \tau \mapsto C, L \wedge \alpha = \tau$, *if* $\alpha \in FV(C)$

S3. $C, L \wedge \alpha \sqsubseteq_j \tau \wedge \alpha \sqsubseteq_j \sigma \mapsto C, L \wedge \tau = \sigma \wedge \alpha \sqsubseteq_j \tau)$

S4. $C, L \wedge \vec{\tau}\kappa \sqsubseteq_j \alpha$
$\quad \mapsto FAIL$ *if* $(\alpha_j \sqsubseteq_{i_j} \alpha_{j+1})_{1 \leq j \leq k} \in L$, *where* $\alpha_1 = \alpha$, *and* $\alpha_k \in FV(\vec{\tau})$
$\quad \mapsto C[\vec{\tau'}\kappa/\alpha], L[\vec{\tau'}\kappa/\alpha] \wedge \alpha = \vec{\tau'}\kappa \wedge \vec{\beta} \sqsubseteq_j \vec{\beta'}$, *where* $\vec{\tau'}$ *is copied from* $\vec{\tau}$,
replacing the free-variables $\vec{\beta}$ *with fresh variables* $\vec{\beta'}$ *otherwise*

$U1-5$ are the well-known algorithm for computing the most general unifier of a set of equations, including an 'occurs check'. $S1-4$ concern the inequations, and the side condition in $S4$ will be referred to as the 'extended occurs check'. A system is said to be in solved form when no rewrite applies to it.

[6] We give a version of Henglein's rewrite procedure, though that due to Leiss is equivalent.

Lemma 1 (Henglein) . *Let (C, L) be in solved form. Define $U := \{\tau/\alpha : \alpha = \tau \in L\}$ and for $1 \leq j \leq n$, $T_j := \{\tau/\alpha : \alpha \sqsubseteq_j \tau \in L\}$. Then $(U, T_1, \ldots, T_n)$ is a most general solution of (C, L)*

Example Consider the system $(\{\}, L1)$ below, consisting of the instantiation claims obtained in phase 1 of typing the `collect`-function of example 5. It can be rewritten to $(\{\}, L10)$, which can then be rewritten no further:

$$
\begin{array}{lll}
 & (\{\}, L1) = & \{\}, \alpha\ T{\to}\alpha\ list \sqsubseteq_1 \alpha\ T\ T{\to}\beta\ list, \\
 & & \quad \alpha\ T{\to}\alpha\ list \sqsubseteq_2 \beta{\to}\alpha\ list \\
\\
\mapsto_{S1} & (\{\}, L2) = & \{\}, \alpha\ T \sqsubseteq_1 \alpha\ T\ T, \\
 & & \quad \alpha\ list \sqsubseteq_1 \beta\ list, \\
 & & \quad \alpha\ T{\to}\alpha\ list \sqsubseteq_2 \beta{\to}\alpha\ list \\
\\
\mapsto_{S1} & (\{\}, L3) = & \{\}, \alpha \sqsubseteq_1 \alpha\ T, \\
 & & \quad \alpha\ list \sqsubseteq_1 \beta\ list, \\
 & & \quad \alpha\ T{\to}\alpha\ list \sqsubseteq_2 \beta{\to}\alpha\ list \\
\\
\mapsto_{S1} & (\{\}, L4) = & \{\}, \alpha \sqsubseteq_1 \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_1 \beta, \\
 & & \quad \alpha\ T{\to}\alpha\ list \sqsubseteq_2 \beta{\to}\alpha\ list \\
\\
\mapsto_{S1} & (\{\}, L5) = & \{\}, \alpha \sqsubseteq_1 \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_1 \beta, \\
 & & \quad \alpha\ T \sqsubseteq_2 \beta \\
 & & \quad \alpha\ list \sqsubseteq_2 \alpha\ list \\
\\
\mapsto_{S1} & (\{\}, L6) = & \{\}, \alpha \sqsubseteq_1 \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_1 \beta, \\
 & & \quad \alpha\ T \sqsubseteq_2 \beta, \\
 & & \quad \alpha \sqsubseteq_2 \alpha \\
\\
\mapsto_{S3} & (\{\}, L7) = & \{\}, \beta = \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_1 \beta, \\
 & & \quad \alpha\ T \sqsubseteq_2 \beta, \\
 & & \quad \alpha \sqsubseteq_2 \alpha \\
\\
\mapsto_{U3} & (\{\}, L8) = & \{\}, \beta = \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_1 \alpha\ T, \\
 & & \quad \alpha\ T \sqsubseteq_2 \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_2 \alpha \\
\\
\mapsto_{S1} & (\{\}, L9) = & \{\}, \beta = \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_1 \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_2 \alpha, \\
 & & \quad \alpha \sqsubseteq_2 \alpha \\
\\
\mapsto_{S3,U4} & (\{\}, L10) = & \{\}, \beta = \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_1 \alpha\ T, \\
 & & \quad \alpha \sqsubseteq_2 \alpha
\end{array}
$$

From the equation in $(\{\}, L10)$ we obtain the substitution $U = [\alpha\ T/\beta]$, and from the inequations we obtain matching substitutions $T_1 = [\alpha\ T/\alpha]$, $T_2 = [\alpha/\alpha]$. (U, T_1, T_2) is a most general solution of $(\{\}, L1)$ and $(\{\}, L10)$.

Example Recall example 5 was a variation on the `collect` example, in which the argument of `collect` had to have the same type as a wider-scoped λ-bound variable. Phase 1 of typing the `collect` declaration will give the system $(\{\alpha\}, L1)$, below. This is rewritten to $FAIL$.

$$
\begin{array}{lll}
 & (\{\alpha\}, L1) = & \{\alpha\},\ \alpha\ T{\rightarrow}\alpha\ list \sqsubseteq_1 \alpha\ T\ T{\rightarrow}\beta\ list, \\
 & & \quad \alpha\ T{\rightarrow}\alpha\ list \sqsubseteq_2 \beta{\rightarrow}\alpha\ list \\
 & & \\
\mapsto_{S1} & (\{\alpha\}, L2) = & \{\alpha\},\ \alpha\ T \sqsubseteq_1 \alpha\ T\ T, \\
 & & \quad \alpha\ list \sqsubseteq_1 \beta\ list, \\
 & & \quad \alpha\ T{\rightarrow}\alpha\ list \sqsubseteq_2 \beta{\rightarrow}\alpha\ list \\
 & & \\
\mapsto_{S1} & (\{\alpha\}, L3) = & \{\alpha\},\ \alpha \sqsubseteq_1 \alpha\ T, \\
 & & \quad \alpha\ list \sqsubseteq_1 \beta\ list, \\
 & & \quad \alpha\ T{\rightarrow}\alpha\ list \sqsubseteq_2 \beta{\rightarrow}\alpha\ list \\
 & & \\
\mapsto_{S2} & (\{\alpha\}, L3) = & \{\alpha\},\ \alpha = \alpha\ T, \\
 & & \quad \alpha\ list \sqsubseteq_1 \beta\ list, \\
 & & \quad \alpha\ T{\rightarrow}\alpha\ list \sqsubseteq_2 \beta{\rightarrow}\alpha\ list \\
 & & \\
\mapsto_{U3} & FAIL &
\end{array}
$$

Lemma 2 (Henglein) *If a problem (C, L) has a solution, then the rewrite system will terminate when applied to (C, L).*

Lemma 1 and 2 entail the following theorem:

Theorem 3 (Henglein) *If a problem has a solution, it has a most general solution.*

The ML case of the principal types theorem is a consequence of the existence of most general unifiers, and in a similar way, the principal types theorem for ML^+ is a consequence of the existence of most general semiunifiers. The algorithm for calculating semiunifiers differs, however, from that for calculating unifiers in the respect that when applied to an unsolvable problem, it may not terminate. This is a consequence of a result due to [Kfoury, Tiuryn, and Urcyczyn, 1989] that semiunification is *undecidable*. This said, the proof of Kfoury et al generates in no very direct way an input to the algorithm on which it will not terminate, and in fact no example has been found on which the algorithm does not terminate. This was one reason for creating the implementation: to enable one to investigate whether or not realistic programming can lead to a typing problem on which the semiunification algorithm will diverge.

2.3 The type-checking algorithm

A Damas-Milner style type-checking algorithm for ML consists of two mutually recursively defined functions, $\mathcal{W}_{exp}$, an `exp`-ression type checker, and $\mathcal{W}_{dec}$, a `dec`-laration type checker:

$$
\begin{array}{l}
\mathcal{W}_{exp}(E, \texttt{exp}) = (S, \tau) \\
\mathcal{W}_{dec}(E, \texttt{dec}) = (S, x \mapsto \tau)
\end{array}
$$

In addition to the output type τ (or environment $x \mapsto \tau$), a substitution S is also produced, which defines a minimum *refinement* of the enviroment which may be necessary to type the code. Concerning $\mathcal{W}_{exp}$ one can show:

(a) If $\mathcal{W}_{exp}(E, e) = (S, \tau)$, then the following are equivalent for all (S', τ')
(i) ML proves $S'E|\!-e : \tau'$ for some refinement S'
(ii) $(S', \tau') = (TS, T\tau)$ for some substitution T.

(b) if $\mathcal{W}_{exp}(E, e) = fail$, then for no (S, τ) does ML prove $SE|\!-e : \tau$

(c) $\mathcal{W}$ terminates on every input.

Hence, $\mathcal{W}_{exp}(E, e)$ fails or computes a *most general* type of e modulo E, and so does $\mathcal{W}_{dec}$ for declarations.

In defining $\mathcal{W}^+$, we have to split E into a part Δ containing assumptions arising from a λ– (or `fn`) binding, and a part Γ for the remainder. As indicated in section 2.1, an additional output parameter, is required, consisting of semiunification problem, L. This will recording the instantiations that have been required for the type variables not occurring in Δ:

$\mathcal{W}^+{}_{exp}(\Delta, \Gamma, \texttt{exp}) = (L, S, \tau)$
$\mathcal{W}^+{}_{dec}(\Delta, \Gamma, \texttt{dec}) = (L, S, x \mapsto \tau)$

$\mathcal{W}^+$ as defined below can be shown to have analogues of the above properties of $\mathcal{W}$:

(a) If $\mathcal{W}^+{}_{exp}(\Delta, \Gamma, e) = (L, S, \tau)$, then $\overline{L}^{S\Delta}$ holds and the following are equivalent for all (S', τ'):
(i) ML^+ proves $S'\Delta, \overline{S'\Gamma}^{S'\Delta}|\!-e : \tau'$
(ii) $(S', \tau') = (TS, T\tau)$, for some T, such that $\overline{TL}^{TS\Delta}$ holds

(b) if $\mathcal{W}_{exp}(\Delta, \Gamma, e) = fail$, then for no (S, τ) does ML prove $S\Delta, \overline{S\Gamma}^{S\Delta}|\!-e : \tau$

(c) if for some (S, τ) ML proves $S\Delta, \overline{S\Gamma}^{S\Delta}|\!-e : \tau$, then $\mathcal{W}^+{}_{exp}(\Delta, \Gamma, e)$ terminates.

A similar statement holds of $\mathcal{W}^+{}_{dec}$.

The algorithm $\mathcal{W}^+$is defined below, in which we use $spec_\Gamma(L)$ ('the free variables of specialisations in L with respect to Γ'), where L is a true set of $\sqsubseteq_{FV(\Delta)}$ claims, to refer to the transitive closure of the set of free variables of Γ under the relation $\alpha R \beta$, where $\alpha R \beta$ holds if if for some $l \sqsubseteq r$ of L, α is a pattern variable of l, and $\beta \in T\alpha$, where T is the associated match.

- $\mathcal{W}^+{}_{exp}(\Delta, \Gamma, x) = (L, \emptyset, \tau[\vec{\beta}'/\vec{\beta}, \vec{\alpha'}/\vec{\alpha}])$
  where $\forall\vec{\beta}\tau = (\Delta, \Gamma)(x)$,
  $\vec{\alpha} = FV(\forall\vec{\beta}\tau) - FV(\Delta)$,
  $\vec{\alpha'}$ = new copies of $\vec{\alpha}$,
  $\vec{\beta}'$ = new copies of $\vec{\beta}$,
  $L = \{\alpha \sqsubseteq_i \alpha' : \alpha \in \vec{\alpha}, \alpha' \in \vec{\alpha'}$ is the corresponding copy of $\alpha\}$, i is an integer indexing no other inequations

- $\mathcal{W}^+{}_{exp}(\Delta, \Gamma, \texttt{e}_1\ \texttt{e}_2) = (S_3S_2L_1 + S_3L_2,\ S_3S_2S_1,\ S_3(\alpha))$
  if $(L_1, S_1, \tau) = \mathcal{W}^+{}_{exp}(\Delta, \Gamma, \texttt{e}_1)$
  $(L_2, S_2, \tau') = \mathcal{W}^+{}_{exp}(S_1\Delta, S_1\Gamma, \texttt{e}_2)$
  $S_3 = mgu(S_2\tau, \tau' \rightarrow \alpha)$ (α fresh)

- $\mathcal{W}^+_{exp}(\Delta, \Gamma, \texttt{fn x => e}) = (L_1, S_1, S_1\alpha{\rightarrow}\tau)$
  if $(L_1, S_1, \tau) = \mathcal{W}^+_{exp}(\Delta + \{\texttt{x} \mapsto \alpha\}, \Gamma, \texttt{e})$, where α is fresh

- $\mathcal{W}^+_{exp}(\Delta, \Gamma, \texttt{let dec in e end}) = (S_2L_1 + L_2, S_2S_1, \rho)$
  if $(L_1, S_1, \{\text{x} \mapsto \overline{\tau}\}) = \mathcal{W}^+_{dec}(\Delta, \Gamma, \texttt{dec})$
  $(L_2, S_2, \rho) = \mathcal{W}^+_{exp}((S_1\Delta, S_1\Gamma + \{\text{x} \mapsto \overline{\tau}\}), \texttt{e})$

- $\mathcal{W}^+_{dec}(\Delta, \Gamma, \texttt{val x = e}) = (L_1, S, \{\texttt{x} \mapsto \forall\vec{\beta}\tau\ \})$
  if $(L_1, S, \tau) = \mathcal{W}^+_{exp}(\Delta, \Gamma, \texttt{e})$
  $\vec{\beta} = FV(\tau) - FV(S\Delta) - spec_{S\Gamma}(L)$

- $\mathcal{W}^+_{dec}(\ \Delta, \Gamma, \texttt{val rec f = e}) = (S_3S_2L_1, R, \{\ \texttt{f} \mapsto \forall\vec{\beta}S_3\tau\ \})$
  if $(L_1, S_1, \tau) = \mathcal{W}^+_{exp}(\Delta, \Gamma + \{\texttt{f} \mapsto \alpha\}, \texttt{e})$
  $S_2 = \text{mgu}(\alpha, \tau)$
  $S_3 = \text{mgsu}(FV(S_2S_1\Delta), (S_2L_1))$
  $R = S_3S_2S_1$
  $\vec{\beta} = FV(S_3\tau) - FV(R\Delta) - spec_{R\Gamma}(S_3S_2L_1)$

- $\mathcal{W}^+_{dec}\ (\Delta, \Gamma, \texttt{datatype}\ \vec{\alpha}tn = c_1\ \texttt{|}\ \ldots c_m\ \texttt{|}\ c_{m+1}\ \texttt{of}\ \tau_{m+1}\ \texttt{|}\ \ldots c_{m+n}\ \texttt{of}\ \tau_{m+n}) = ([], \emptyset, E_2)$
  where $\{\text{tn} \mapsto \kappa\}$ is first extended to E_1 by $\{c_i \mapsto \forall\vec{\alpha}.\vec{\alpha}\kappa\}$, for $1 \leq i \leq m$,
  and then E_1 is extended to E_2 by $\{c_i \mapsto \forall\vec{\alpha}.\tau_i[\kappa/tn]{\rightarrow}\vec{\alpha}\kappa\}$, for $m+1 \leq i \leq m+n$

- $\mathcal{W}^+_{dec}$ and $\mathcal{W}^+_{exp}$ return $fail$ if in any of the above cases the called functions return fail

The important points to note are:

1. The standard algorithm $\mathcal{W}$ can be obtained if all variables in the environment are treated as Δ (i.e. λ-bound) variables.

2. The base case contrast with $\mathcal{W}$ is that type variables not occurring in the Δ part of the environment are *copied* in typing an atomic piece of syntax, the copying being reflected in an output *semiunification* problem. Note the issue 'copy or not copy' arises not just for recursively bound variables.

3. With the exception of the case for recursive declarations, the output semiunification problem for complex code is accumulated in the obvious way from the semiunification problems for the component parts.

4. Besides the base case, the other main difference to $\mathcal{W}$ is in the case of recursive declarations, where there is a call to $mgsu$ – a most general semiunifier function – on a semiunification problem defined by typing the definition. The output semiunification problem for the declaration has had the $mgsu$ applied to it.

5. In both non-recursive and recursive declarations the criterion controlling generalisation of type variables involves *both* the environment *and* semiunification problem.

The following examples illustrate points 2 and 5.

Example: *incomplete with copying only of 'rec-bound'*

In the base case a particular 'copying' criterion is used to determine which type variables should be copied – namely those not occurring in the Δ part of the environment. The simple examples of section 1.2 may have suggested an alternative copying criterion – namely copy those occurring in an assumption concerning a recursively bound term-variable. However, the recursively bound variable may occur in the definition part of different `let`-declared variable. Clearly the further occurrences of this `let`-declared variable should count somehow as contributing further instantiation constraints on the polymorphic assumption for the recursively bound variable. Consider for example, the following notational variant of our earlier example 1.

```
val rec f = fn x => let val g = f in (g 1; g ''a''; x)
```

With just the copying of 'rec-bound' variables, the declaration would get rejected. Calling $\mathcal{W}^+$ on this in an empty context, will lead to a call on the embedded `let` expression, in a context $\{x:\gamma, f:\alpha\}$. We show below how this calculation unfolds:

$$\dfrac{\dfrac{\mathcal{W}^+(\{x:\gamma, f:\alpha\}, \texttt{f}) = (\alpha \sqsubseteq \alpha_1, \emptyset, \alpha_1)}{\mathcal{W}^+(\{x:\gamma, f:\alpha\}, \texttt{val g = f}) = (\alpha \sqsubseteq \alpha_1, \emptyset, g:\alpha_1)} \quad \dfrac{\begin{array}{l}\mathcal{W}^+(\{x:\gamma, f:\alpha, g:\alpha_1\}, \texttt{g1}) = ([], \{int{\rightarrow}\alpha_2/\alpha_1\}, \alpha_2) \\ \mathcal{W}^+(\{x:\gamma, f:\alpha, g:int{\rightarrow}\alpha_2\}, \texttt{g ''a''}) = fail\end{array}}{\mathcal{W}^+(\{x:\gamma, f:\alpha, g:\alpha_1\}, \texttt{g 1; g''a''; x}) = fail}}{\mathcal{W}^+(\{x:\gamma, f:\alpha\}, \texttt{let val g = f in ...}) = fail}$$

First the declaration part of the `let` is typed, `val g = f`, which in turn requires the typing of the defining expression, `f`. This occurrence of the recursively bound variable is typed as α_1 in the top-left call in the above, associated with the inequality $\alpha \sqsubseteq \alpha_1$. From this type for the defining expression, a type for the declaration `val g = f` is obtained by quantifying the $\vec{\beta}$ which are $FV(\alpha_1) - FV(x:\gamma) - spec_{f:\alpha}(\alpha \sqsubseteq \alpha_1)$. In this case this rules out the quantification of α_1. Then the `in` part of the `let` is typed, in a context with the assumption `g`:α_1. Then *if* we had copying only for 'rec-bound' variables, the occurrences of `g` must be typed without copying. Then after `g 1` is typed, `g ''a''` would be typed in context with the assumption `g`:$int{\rightarrow}\alpha_2$, and this would lead to $fail$.

With the copying criterion actually used by $\mathcal{W}^+$, the example is typable. However, it would also be possible to derive this example by changing the *quantification* criterion, and persisting with copying only of 'rec-bound' variables. The usual criterion for quantifying a type-variable in ML is that the type-variables not occur in the environment, and if we had this criterion, then `g` would be declared as having type $\forall\alpha_1.\alpha_1$, and the above example could be typed. The following example shows, however, that this quantification criterion is too generous.

Example: *quantifying variables not in environment is unsound*

With a less strict quantification criterion, the following would be typed as $\{\texttt{f} \mapsto int * int{\rightarrow}int\}$, when it is in fact ML^+-*untypable*.

```
val rec f = fn x => let val g = f in (g 1; x = (1,2); 1)
```

As before calling $\mathcal{W}^+$ on this will lead to a call on the embedded `let`, in a context $\{x:\gamma, f:\alpha\}$. The computation on the declaration part then is – *assuming a quantification criterion that checks simply for the presence of a free variable in the environment*:

$$\dfrac{\mathcal{W}^+(\{x:\gamma, f:\alpha\}, \texttt{f}) = (\alpha \sqsubseteq \alpha_1, \emptyset, \alpha_1)}{\mathcal{W}^+(\{x:\gamma, f:\alpha\}, \texttt{val g = f}) = (\alpha \sqsubseteq \alpha_1, \emptyset, \texttt{g}:\forall\alpha_1.\alpha_1)}$$

In the enviroment associated to the declaration, `g` is typed as $\forall\alpha_1.\alpha_1$. The calculation on the `in` part of the `let` is then:

$$\frac{\begin{array}{rcl}\mathcal{W}^+(\{x:\gamma, f:\alpha, g:\forall\alpha_1.\alpha_1\},\ \texttt{g 1}) & = & ([], \emptyset, \alpha_2)\\ \mathcal{W}^+(\{x:\gamma, f:\alpha, g:\forall\alpha_1.\alpha_1\},\ \texttt{x = (1,2)}) & = & ([], int*int/\gamma, bool)\\ \mathcal{W}^+(\{x:int*int, f:\alpha, g:\forall\alpha_1.\alpha_1\},\ \texttt{1}) & = & ([], \emptyset, int)\end{array}}{\mathcal{W}^+(\{x:\gamma, f:\alpha, g:\forall\alpha_1.\alpha_1\},\ \texttt{(g 1; x = (1,2); 1)}) = ([], int*int/\gamma, int)}$$

Thus for the `let` expression we have:

$$\mathcal{W}^+(\{x:\gamma, f:\alpha\},\ \texttt{let ...}) = ([\alpha \sqsubseteq \alpha_1, int*int/\gamma, int)$$

and in typing the recursive function, we will calculate

$$\begin{array}{l} mgsu([\alpha \sqsubseteq \alpha_1][int*int{\rightarrow}int/\alpha])\\ = mgsu([int*int{\rightarrow}int \sqsubseteq \alpha_1])\\ = int*int{\rightarrow}int/\alpha_1\end{array}$$

and thereby, the recursive function will be declared with type $int*int{\rightarrow}int$. With the more restrictive quantification criterion given in the definition of $\mathcal{W}^+$, we will have a different semiunification problem, namely

$$\begin{array}{l} mgsu([\alpha \sqsubseteq \alpha_1, \alpha_1 \sqsubseteq int{\rightarrow}\alpha_2][int*int{\rightarrow}int/\alpha])\\ = mgsu([int*int{\rightarrow}int \sqsubseteq \alpha_1, \alpha_1 \sqsubseteq int{\rightarrow}\alpha_2])\\ = fail\end{array}$$

3 Prototype of a polyrec checker in SMLofNJ

In the previous section we have formulated the typechecker of ML^+ as a close variant of the typechecker for ML as it appears in [Damas and Milner, 1982]. In the `polyrec_sml` implementation, for practical reasons, we remain as close as possible to the typechecker part of the SMLofNJ compiler. Now, as we come to describe the implementation in more detail, some differences between theory and practice in the SMLofNJ typechecker have to be pointed out.

First, recall that the Damas-Milner algorithm uses an extension of the environment to record the interim type-assumptions made concerning the bound variables of a term, and then uses this

- to type occurrences of the bound variable (base case), and
- to control generalisation of type variables (declaration cases).

The typechecker in the SMLofNJ compiler DOES NOT not use an *environment* to make interim type-assumptions concerning bound variables, and accomplishes the above central tasks differently:

- types are injected into the code to type the occurrences of bound variables (see 3.1)
- quantification of type variables is achieved by *depth controlled generalisation* (see 3.1)

Second, instead of storing *substitutions* to be applied later, the substitution is immediately cashed out. These differences between theory and practice are inherited by our implementation of ML^+ typechecking. In the following section we define what might be called a prototype version of our implementation, closing principally these gaps between theory and practice. The hope is that thereby the outline will be clear.

The third difference between the theory described above and the implementation, is that the implementation concerns a larger language (including records, user-defined variables, reference values etc.) and deals with a particular representation of that larger language. Documentation of the implementation proper begins in section 4.

3.1 The representation of terms and types in SMLofNJ

Instead of using an environment to give types to occurrence of bound variables, in SMLofNJ, the occurrences of the variables bear tags which are pointers to a type. That is to say, the tags have type `ty ref`, where `ty` is the datatype for types. One of the values in this type is `UNDEFty`, and initially the tags are references to this value. The tagging proceeds roughly as follows:

`fn x => (... x ...)` $\rightsquigarrow$ `fn` x^m `=> (...` x^m `...)`	:a λ-abstracted variable, and the occurrences in its scope, receive the same m, a fresh reference to `UNDEFty`
`val rec x = (... x ...)` $\rightsquigarrow$ `val rec` x^m `= (...` x^m `...)`	:a recursively declared variable, and the occurrences in its definition, receive the same m, a fresh reference to `UNDEFty`
`val x = (...)` $\rightsquigarrow$ `val` x^m `= (...)`	: a non-recursively declared variable receives m, a fresh reference to `UNDEFty`
`let dec[x] in (... x ...)` $\rightsquigarrow$ `let dec[`x^m`] in (...` x^m `...)`	: the occurrences of a let bound variable receive the same tag as the variable receives in the declaration part.

Type checking proper ensues as a process of revising the tagged code, so that the type references are updated to references to appropriate types.

Types are values in the datatype `ty`. We give an extract of this below, followed by some remarks concerning this representation of the types.

```
ty
  = VARty of tvinfo ref
  | CONty of tycon * ty list
  | IBOUND of int
  | WILDCARDty
  | POLYty of {sign: {weakness:int, eq:bool} list, tyfun: tyfun, abs: int}
  | UNDEFty

tvinfo
  = INSTANTIATED of ty
  | OPEN of {depth, weakness, eq, kind: tvkind}
```

```
tvkind = META | ...

tyfun
  = TYFUN of {arity : int, body : ty}
```

References in open type variables

Open type-variables are represented by values which have as an essential part a *reference* value, that is, an address in memory. Thus variables are indexed not by integers but by memory locations. Consider the evaluation of the following code:

```
val v = VARty(ref(OPEN{depth = d, kind = k, eq= e, weakness = w}))
val v' = VARty(ref(OPEN{depth = d, kind = k, eq= e, weakness = w})).
```

The value of `v`, is the encoding of an *open* type-variable. The value has as a subpart a *reference*, m, to a value `OPEN {depth = d, kind = k, eq= e, weakness = w }`, which records various kinds of information about the variable, the most important kind being that it is an *open* variable. The value of `v'` is the encoding of a distinct – though similar – open variable. The value will contain another reference m', distinct from m. The contents of the two references, however, are identical.

The instantiation of type-variables is carried out by updating the contents of the reference part. For example, to instantiate `v` to say *int*, one could evaluates:

```
let val VARty(m1) = v in (m1 := INSTANTIATE(int)) end
```

The contents of the reference part of `v` are thereby changed to `INSTANTIATE(int)`. This will cause the appropriate effect also when `v` is embedded, as in `v --> v'`, where `-->` denotes the arrow type constructor. Thus no substitution operation has to be written, which is the great advantage of this representation of type-variables. The cost is that the result of 'instantiating' the variable `v` to *int* is not simply *int* but `VARty(ref(INSTANTIATE(`*int*`)))`. In fact there are infinitely many values in the `ty` datatype which effectively represent *int*, of the form `VARty(ref(INSTANTIATE(...(VARty(ref(INSTANTIATE(`*int*`)))))))`, and certain parts of the implementation have to take care of these equivalences.

Quantified types: the representation maintains a distinction between bound and free type variables. `IBOUND(`n`)`, for integer n, represents the various bound variables, and these are intended to occur in the scope of the `POLYty` operator. Contrary to the usual definiton of ML types, this representation of types does allow types with embedded quantified types. These, however, never arise in the course of typechecking.

The `weakness` and `eq` fields that appear in open variables do not appear in the occurrences of bound variables, but this information is still associated with the variables by the `sign` field of a quantified type.

Depths and depth-controlled generalisation

Because the compiler does have an environment for interim type assumptions, the quantification of type-variables cannot be controlled in the way suggested directly by the theory. Instead `depth`-controlled generalisation is used.

- A type variable has `depth` d if the *widest* scoped abstraction with which it is associated is embedded at a depth $d-1$ under further abstractions. For example, if `fn` $\mathtt{x}^m$ `=>` $\mathtt{x}^m$ is well typed then,

```
!m = VARty(ref(OPEN{depth = 1,eq=false,weakness=infinity,kind=META }))
```

- Depths are adjusted in unification: when a variable of depth d is equated with a type which features variables of depths $d_1, ...d_n$, each of the d_i is adjusted to $minimum(d, d_i)$.
- The typecheck functions have (essentially) a `depth` argument, which records the number of `fn x =>` and `val rec` bindings that have been descended through.
- In generalisation: type variables with `depth` attribute $>$ 'current' depth may be generalised.

3.2 The prototype checker

The parsing stage gives values of type `Ast.dec` - for the purposes of this outline, this we can equate with the ML declarations defined earlier. Note the checker always starts on a declaration. These are then 'tagged' in the way described above. The tagging of the trees (of type `Ast.dec`) takes them into the datatype of `Absyn.dec` trees.

The main auxiliary functions/values used by the prototype typechecker are described below. Those marked + are additions to what it required for ML:

	`instantiate`	`:ty -> ty` takes a quantified type and replaces the bound variables with depth infinity open variables.
+	`rulebound`	`:ty list ref` reference to the list of open type variables occurring in currently in scope λ-bound variables
+	`copy`	`:ty list ref -> ty -> ty` variables not on the rule-bound list get copied (with depth preservation), and semiprob is updated. Depth infinity variables resulting from the initial instantiations of a bound variable are *not* copied
	`unifyTy`	`:ty * ty -> unit` This is a side-effecting operation which basically makes its arguments equal. There is a further important aspect concerning depths: when a variable of depth d is equated with a type which features variables of depths $d_1, ...d_n$, each of the d_i is adjusted to $minimum(d, d_i)$.
+	`Semiunify.ineq`	`: ineq` The datatype of integer indexed inequations
+	`Semiunify.process`	`:ineq list -> ineq list` This basically applies the earlier defined rewrites S1-S4, taking rule-bound as the context set. However, when S1-S4 specify that an identity, $\alpha = \tau$, be added to the list, here α is simply instantiated to τ.
	`generalize`	`:ty ref -> unit` takes a depth d, and a type reference, and updates the reference to a type in which open type variables of depth $> d$ are quantified.

Below we give a prototype of the polyrec checker. With certain obvious deletions, that the comments

below should make clear, this can also be read as a summary version of the existing SMLofNJ typechecker.

```
expType(d,x^m) =
   let val ty1 = instantiate(!m)
       val ty = copy(rulebound, ty1) (* not in original *)
    in (x^m,ty)                      (* output is ty1 in original *)
   end

expType(d,e1 e2) =
   let (e1',fty) = exptype(d,e1)
       (e_2',aty) = exptype(d,e2)
       resty = VARty(ref(OPEN{depth = infinity,...}))
    in (unify(aty -> resty,fty);
       (e1'e2',resty))
   end

expType(d,fn x^m => e) =
   m := VARty(ref(OPEN{depth = d+1,...}));
   let val currentbound = (!rulebound)           (* not in original *)
       val _ = rulebound := (!m) :: (!rulebound) (* not in original *)
       val (e',ety) = expType(d+1, e)
    in (rulebound := currentbound;               (* not in original *)
        fn x^m => e', !m -> ety)
   end

expType(d, let dec in e) =
   let dec' = decType(d, dec)
       (e',ety) = expType(d, e)
    in (let dec' in e', ety)
   end

and

decType(d, val x^m = e) =
   let val _ = m := VARty(ref(OPEN{depth = infinity,...};
       val (e',ety) = expType(d, e)
       val _ = unifyTy(!m,ety)
       val _ = generalize(d, m)
    in (val x^m = e)
   end

decType(d, val rec x^m = e) =
   let val _  m := VARty(ref(OPEN{depth = d+1,...}))
       val (e,ety) = expType(d+1, e)
       val _  = unifyTy(!m,ety)
       val _ = semiprob := Semiunify.process(!semiprob) (* not in original *)
       val _ = generalize(d,m)
    in (val rec x^m = e)
   end
```

Salient differences to the typecheck function $\mathcal{W}^+$ defined earlier:

- For none of the typecheck functions is there an input environment, nor an output substitution. Instead there are side-effects to type references embedded in the code.
- There is side-effectable reference to a list of types, `rulebound` which contains the free variables of currently in scope λ-variables.
- Copying of type variables is controlled by the contents of the `rulebound` list
- None of the typecheck functions output a semiunification problem. There is instead a side-effectable reference to a global semiunification problem.
- Generalisation of types is depth-controlled, as it is in SMLofNJ, but not as in $\mathcal{W}^+$

The following section gives detailed documentation of the `polyrec_sml` implementation.

4 The polyrec checker

4.1 Summary of differences

The main changes are the addition of the code defining a structure `Semiunify`, and the modification of the code defining the structure `Typecheck`. Also:

1. `polyrec_prev.sml` differs from `perv.sml` by
   System.Print:
   ```
   < val printaftergen = ref false
   ```
   System.Control.CG:
   ```
   < val oldchecker = ref false
   ```
   System.Control.CG:
   ```
   < val viewsemi = ref false
   ```
   System.Control:
   ```
   < val primaryPrompt = ref "+ "
   ---
   > val primaryPrompt = ref "- "
   ```
   System.Control:
   ```
   < val usemono = fn () => (InLine.:=(CG.oldchecker,true);
   <                         InLine.:=(primaryPrompt,"- "))
   < val usepoly = fn () => (InLine.:=(CG.oldchecker,false);
   <                         InLine.:=(primaryPrompt,"+ "))
   ```

2. corresponding differences between `polyrec/polyrec_ system.sig` and `boot/system.sig`
   signature CGCONTROL
   ```
   < val oldchecker:  bool ref
   < val viewsemi:  bool ref
   ```
   signature PRINTCONTROL
   ```
   < val printaftergen :  bool ref
   ```
   signature CONTROL
   ```
   < val usemono :  unit -> unit
   < val usepoly :  unit -> unit
   ```

3. `polyrec/polyrec_batch.sml`: the `flags` list in `Batch` functor has extra member `("oldchecker",oldchecker)`, so that batch compilation can proceed both with and without polymorphic recursion

4. `polyrec/polyrec_pptypelist.sml`: defines an additional pretty printer structure `PPTypelist`, containing `ppineqlist` and `pptypelist` for pretty printing a semiunification problems and a list of types. `polyrec/pptype.sml` is exactly the same as the standard `print/pptype.sml` code, except the signature includes the function `ppType1`.

4.2 Documentation

The first entry indicates whether this code is additional (+), or a modification (∼). Documentation is also included of some the unaltered code. We use the same conventions as the pretty-printing functions for variables - 'A,'B,'C etc for unbound, 'a,'b,'c for bound.

1. ∼ `Typecheck : the Typecheck structure`

   Original structure contained a definition of the form:

```
dectype(env,dec,toplev,err,loc) =
let
fun generalizeTy ...
fun generalizePat ...
fun applyType ...
fun patType ...
fun expType(....) and ... and decType0(...)
val _ = resetOverloaded()
val dec' = decType0(dec,...)
val _ = resolveOverloaded
in dec'
end
```

   The new version of `Typecheck` contains a definition of the form:

```
dectype(dec,....) =
if System.Control.CG.oldchecker = true then
(let
fun generalizeTy
fun generalizePat
fun applyType
fun patType
fun expType(....) and ... and decType0(...)
val _ = resetOverloaded()
val dec' = decType0(dec,...)
val _ = resolveOverloaded
in dec'
end)  - all as before

else
(let
```

```
fun generalizeTy
fun generalizePat
fun applyType
fun patType
fun expType(....) and ... and decType0(...) - modfied a bit
val _ = reset0verloaded()
val dec' = decType0(dec,...)
val _ = resolve0verloaded
in dec'
end)
```

The 'newchecker' branch has additional values and modifications of previous values

2. + `typprint` : `ty -> unit`

   Prints a type. See [22].

3. + `nameprint` : `symbol -> unit`

   Prints a variable name. See [22].

4. + `inpr` : `Semiunify.ineq list -> unit`

   Prints a semiunification problem. See [7].

5. + `lpr` : `ty list -> unit`

   Prints a list of types. See [7]

6. + `rvbpr` : `Absyn.rvb -> unit`

   Prints a recursive value binding. See [8]

7. + `checkpr` : `Semiunify.ineq list -> string -> Semiunify.ineq list`

   See `Semiunify.process`[45], `Semiunify.show` [42]. After a semiunification rewrite step, is called on the resulting semiunification problem, and a string giving the name of the rewrite. Always returns the semiunification problem, after possibly printing some output. Used to allow – according to the value of
   `System.Control.CG.viewsemi` – optional feedback after a semiunification rewrite step. If `System.Control.CG.viewsemi` is true, user is prompted with `print ?`, and if the answer is `y`, the name of the rewrite rule is printed, the `rulebound` variables, and the resulting semiunification problem. Otherwise the semiunification problem is returned.

8. + `checkrvbpr` : `Absyn.rvb -> unit`

   According to the value of `System.Control.CG.viewsemi`, gives the user the option to see the recursive value binding about to be typechecked. See [6]

9. + `rulebound` : `ty list ref`

   See `copyTy` [18], `copy` [19], `Semiunify.Process` [45], `ruleType` [30], `rbupdate` [17]. This is a reference to the list of types of currently in scope rule-bound variables. Used to control copying of variables and the rewriting of the `semiprob` semiunification problem

   The reference is actually created in the structure `Semiunify`, as the value of the variable `Semiunify.nonpat`. Within the `Typecheck` structure, the value of the variable is defined to be that of `Semiunify.nonpat`

10. + topprob : boolref

    See VALRECdec case of decType0 [32]. This is a reference to a boolean used to record whether a whether a top-level recursive declaration is being checked or an embedded

11. + semiprob : ineq list ref

    Reference to a semiunification problem, which a list of terms ineq(int,ty,ty). See copy [19], VALRECdec case of decType0 [32]

12. + eqcounter : int ref

    See copy [19]. This is a reference to an integer, which is the inequality counter, used to make sure that when an inequality is added to the semiprob list, it receives a new indexing integer

13. + member : 'a -> 'a list -> bool

    Membership function on any list. Defined in structure Semiunify.

14. + there : ty -> ty list -> bool

    Membership function on lists of types, treating prune-equal types as equal. See prune [20]

15. + readrulebound : ty list ref -> unit

    A variable in Typecheck structure which is identified with a function of the same name defined in Semiunify. See copy [19], Semiunify.reduce1a [47].

    Takes a reference to a list of types, and updates the reference to the duplicate free list of the free variables. For example if rulebound refers to ['A -> 'B,'B -> 'C], after the call of readrulebound, this will have been compressed to ['A,'B,'C]

16. FLEX of label * ty list : possible value of kind

    One of the possible values of the kind field in open variables is FLEX of label * ty list. This arises in typing an underspecified record. Consider an underspecified record with two specified fields, {l1 = v1, l2 = v2, ... }, where the values v1 and v2 have types ty1 and ty2. This receives a type of the form:

    ```
    VARty(ref(OPEN {depth=d, eq=q, weakness=w, kind = FLEX([(l1,ty1),(l2,ty2)])
    }))
    ```

    Note this representation of the type of an underspecified record is not isomorphic to the representation used for a fully specified record, such as {l1 = v1, l2 = v2 }, which is

    ```
    CONty(RECORDtyc([l1,l2]),[ty1,ty2])
    ```

    See rbupdate [17], copyTy [18]

17. + rbupdate : ty -> unit

    See ruletype [30]

    Takes the initially assigned type of the argument of a function, prunes it – see [20] – and updates rulebound with the free variables that occur in that type. In more detail:

    (a) in case the type is CONtty(f,args), rbupdate is iterated over args

    (b) in case the type is a flexible record type: VARty(ref(OPEN {kind = FLEX([(l1,t1) ... (ln,tn)]),... })), rbupdate is iterated over the list [t1,...,tn]

(c) in case the type is on open variable, it is added to `rulebound` list.

18. + `copyTy` : `ty -> ty * (ty * ty) list`

see `copy` [19].

Takes a type and returns a certain copy of that type, together with `ty * ty` list which records which variables were copied to what types. This list is later used by `copy` to update the `semiprob`.

The basic idea is that variables not on the `rulebound` list get copied, all occurrences of a particular variable getting the same copy. In addition

(a) user-bound variables not copied.

(b) a flexible record is `VARty(m1)`, where `m1` is `ref(OPEN {kind=FLEX(fields),... })`. This variable is not copied, but the contents of `m1` are updated by recursively applying copying to the unknowns in `fields`. Thus supposing an empty `rulebound` list, the flexible record `{1:'A,2:'B,'...Z }` will be copied to `{1:'C,2:'D,'...Z }`.

(c) depth infinity variables are the initial instantiations -see [25] – of a bound variable, and are not copied

(d) zero-depth variables arise from over-loaded symbols, and are not copied, for which a copying would block resolution.

19. + `copy` : `ty -> ty`

takes a type, uses `copyTy` to copy it, and with resulting record of the copying that was done, updates `semiprob` with appropriate inequations. The inequation counter `eqcounter` is first incremented to ensure the new inequations have a new index.

20. `prune` : `ty -> ty`

Defined in util/Typesutil.sml by:

```
fun prune(VARty(tv as ref(INSTANTIATED ty))) : ty =
      let val pruned = prune ty
       in tv := INSTANTIATED pruned; pruned
      end
  | prune ty = ty
```

This returns exactly `ty` when either `ty` is not a variable type, or if `ty` is an open variable type, i.e. `VARty(ref(OPEN {... }))`. In case `ty` is one the above cases, `ty'`, embedded under several layers of `VARty(ref(INSTANTIATED ...))`, the result is to return `ty'`, and with the side effect that `ty` then has the form `VARty(ref(INSTANTIATE ty'))`

21. `unifyTy(type1,type2)` : `ty * ty -> unit`

Defined in src/basics/unify.sml. `unifyTy` makes a case distinction on `headReduceType(prune type1)` and `headReduceType(prune type1)`. `headReduceType` is defined in src/basics/typesutil.sml, and is redundant unless type abbreviations are involved, in which case these are cashed out. For prune see [20]. `unifyTy` then distinguishes six cases

(a) `WILDCARDty,_` - no side effect

(b) `WILDCARDty,_` - no side effect

(c) `(VARty var1,VARty var2)`, where `var1 = ref(OPEN {d1,e1,w1,k1 })` and `m2 = ref(OPEN {d2,e2,w2,k2 })`.

if `m1 = m2` then `()` else (roughly)

```
m1 := INSTANTIATED VARty(m2)
m2 := OPEN{depth = min(d1,d2),
           eq = e1 orelse e2,
           weakness = min(w1,w2),
           kind = unify_tvinfos(k1,k2)}
```

So `m2` is updated to to refer to variable information whose depth is the minimum of the depths of the two contributing variables, and `m1` is basically instantiated to `m2`. If both `VARty(m2)` and `VARty(m1)` are distinct user-bound variables a `Unify ``bound type var''` exception is raised (by `unify_tvinfos`). Because userbound variables cannot be instantiated, if `m1` is user-bound then instead `m2` is basically instantiated to the adjusted version of `m1` (there are further conditions concerning the weakness and equality attributes of user-bound variables and their instantiations)

When either of `k1` or `k2` is `META`, the result of `unify_tvinfos` is the other one, and if both are `FLEX fields` then the the fields are merged and the types of shared fields are unified.

(d) `(VARty var1, type2)` case. `var1` is `m1 = ref(OPEN {kind=META,depth=d,... })`. Basically the result is

```
m1 := INSTANTIATED type1
```

However first `adjust_type m1 type2` is evaluated. This carries out an occurs check for occurrences of `var 1` in `type2` (and which may raise `Unify ``circularity''` and adjusts variables in `type2` with depth `d'` to have depth `min(d,d')`.

There is also a further case when `VARty var1` is a flexible record type, and `type2` is a record type.

(e) `(type1, VARty var2) = unifyTy(VARty var 2,type1)`

(f) `(tycon1 args1, tycon2 args2)` case. The equality of `tycon1` and `tycon2` are checked (which may raise `Unify ``tycon mismatch''`) then the arguments are pairwise unified.

22. `generalizeTy : var * tyvar list * occ * (linenum * linenum) -> unit`

This is the function used to generalize the type inferred for a variable bound in a recursive declaration – see `decType0` [32]. It also adapted in `generalizePat` – see [23] – to generalize the type inferred for a pattern in a non-recursive declaration – see `decType0` [31]

The function is of the form:

```
fun generalizeTy(VALvar{typ,...}, userbound: tyvar list, occ:occ, loc) : unit =
    let ...
        fun gen(ty) = ...
        val ty = gen(!typ)
     in (typ := POLYty{sign = ..., abs = ...,
                       tyfun = TYFUN{arity= ...,body=ty}};
         if !System.Print.printaftergen    (* this print clause added *)
         then typprint (!typ)
         else ()  )
    end
```

Basically the type reference `typ` is updated to the generalized version of the previous reference. This is accomplished principally by the function `gen`, which replaces particular free variables of `!typ` with bound variables, `IBOUND(n)`. The `weakness` and `eq` attributes of the variables generalized appear in the `sign` field of the resulting type.

The final clause is an addition which simply prints the type obtained after a step of generalization, making the types assigned in embedded declarations visible.

What determines whether a variable may be generalized is the relationship between its attributes and the `occ` value that is the third argument of `generalizeTy`, as defined by `gen(ty)`.

The base case of `gen(ty)` is when `ty` is a variable. Otherwise `gen(CONty(tycon args)) = CONty(tycon,map gen args)`. When `ty` is a variable `VARty(ref(OPEN {depth,weakness,eq,kind }))`, `gen(ty)` makes a three way distinction according to the `kind` of the variable. The cases `kind = META`, `kind = FLEX _` are documented below.

(a) `kind = META`
If the variable's `depth` is greater than `lamdepth occ` (and `weakness > generalize_point occ`) then the variable is replaced by a bound variable. The `sign` field of the eventual quantified type will record the `weakness` and `eq` attributes the generalized variable.

(b) `kind = FLEX _`
The SMLofNJ checker has the property that a flexible record type cannot be generalised - an error message is generated, and a wildcard is given as the answer.

23. `generalizePat : pat * tyvar list * occ -> unit`

Called by `decType0 VALdec` case [31], to generalize particular free variables in the type of a pattern.

24. `applyType : ty * ty -> ty`

Basically `applyType(ty1,ty2)` unifies the argument part of `ty1` with `ty2`, and defines the answer as the value part of `ty2`. See `expType APPexp` case [29]

```
fun applyType(ratorTy,randTy) =
    let val resultType = VARty(ref(OPEN { depth = infinity,
                                          weakness = infinity,
                                          equality = false,
                                          kind = META }))
     in (unifyTy(ratorTy, (randTy --> resultType));resultType)
    end
```

25. `instantiateType(ty,occ) : ty * occ -> ty`

Defined in basics/typesutil.sml. Called in `expType`, `VARexp` case [27]

Roughly, `instantiateType` is an identity on monotypes, and in the case of polytypes, replaces the bound type-variables with 'open' type variables of depth infinity. This infinite depth allows the possibility that the instantiation may be reversed in a generalisation, when the occurrence of the polytyped variable does not constrain the relevant argument place.

Over and above this, though, `instantiateType` also computes new weakness for the argument slots of the body of the polytype. So `instantiateType` returns a modified version of the body of a polytype, replacing the `IBOUND n`'s in the body with 'open' variable types of *infinite*

depth, and appropriate weakness as determined by `calc_weakness(abs,occ,w)` where `w` is the appropriate weakness entry in `sign`, whilst any 'open' variable types in the body are left unchanged, up to change of weakness, again calculated by `calc_weakness(abs,occ,w)` where this time `w` is the weakness of the given open variable type.

`instantiateType(ty,occ)` is defined as follows:

(a) case `ty` is not a polytype `=> ty`

(b) case `ty = POLYty{sign,body,abs} => subst body`.
`subst body` = case body of

i. `VARty(ref(INSTANTIATED(ty))) => subst ty`

ii. `CONty(tyc,args) => CONty(tyc,subst args)`

iii. `VARty(r as ref(OPEN{weakness = w,...})) => VARty(r')`, where `r'` has adjusted weakenss

iv. `IBOUND n => VARty(ref(OPEN{kind = META,depth = infinity, weakness = w,...}))`, where `w` is an adjusted weakness

`calc_weakness` is defined basics/typesutil.sml.

26. `mkRefMETAty : int -> ty`

defined in basics/typesutil.sml.

```
mkRefMETAty(d) =
VARty(ref(OPEN{depth = d,weakness=infinity,eq=false,kind=META }))
```

27. ~ `expType(exp,occ,loc) : case exp of VARexp(r as ref(VALvar {typ,access,name }),_) =>`

```
let val ty1 = instantiateType(!typ,occ)
    val ty = (readrulebound rulebound; copy ty1)
 in if Prim.special access  (*  =, <>, :=, update  special cases *)
    then (r := VALvar{typ= ref ty,access=access,name=name};
                    (VARexp(r,NONE),ty))
    else (case (!typ)
                  of POLYty _ => (VARexp(r,SOME ty),ty)
                   | _ => (VARexp(r,NONE),ty))
end
```

Essentially `VALvar {typ, access, name }`, where `typ` has type `ty ref`, is a bound occurrence of a variable. The representation, however, embeds such a variable, `v`, further, as `VARexp(ref(v),tyop)`, where `tyop` may either be `NONE` or `SOME ty`. The main point is that the output type `ty` is derived from the contents of the type reference, `typ`, by first instantiating [25] and then copying [18]. `copy` will basically copy the type variables not on the `rulebound` list. `readrulebound rulebound` is a compression operation on the `rulebound` list – see [9].

Note that in most cases this call to `expType` is without side-effects. However, in the case of the 'special access' variable, the type obtained by instantiation and copying is used to update the reference `r` in `VARexp(r,_)` to `VALvar {typ = ref ty,...}`.

The `tyop` argument in the output is used to record whether or not the type of the variable was polymorphic before the type-checking.

28. `occ : occ`

An `abstype` defined and documented in *src/basics/typesutil.sml*. A value of type `occ` will consist of the constructor `OCC` applied to a record with various fields, one of which is `lamd` = *lambda depth*. Amongst the functions defined in the `abstype` are `Abstr`, which (amongst other things), increments `lamd` by one, and `lamdepth`, which returns the `lamd` value.

29. `expType(exp,occ,loc) : case exp of APPexp(rator, rand) =>`

```
let val (rator',ratorTy) = expType(rator,Rator occ,loc)
  val (rand',randTy) = expType(rand,Rand occ,loc)
  val exp' = APPexp(rator',rand')
in (exp',applyType(ratorTy,randTy))  handle Unify(mode) => let ....  in (exp,WILDCARDt
end
end
```

This is unchanged from the SMLofNJ compiler. Basically the operator and operand are typed at `occ` levels `Rator occ` and `Rand occ`, where `Rator` and `Rand` are operations defined in the `abstype` for `occ` – see [28] . Note these adjust `weakness` relevant aspects of `occ`. The `lamdepth` attribute is not altered, i.e. `lamdepth(occ)` = `lamdepth (Rator occ)` = `lamdepth (Rand occ)`. The exceptions generated by the unification code are of the form `Unify(mode)`, where `mode` is a string – see [21] – and the exception handling involves printing various error messages and then defining the answer as `(exp, WILDCARDty)`. Defining `WILDCARDty` as the answer allows typechecking to continue after some error has been discovered, allowing potentially further errors to be detected.

30. ∼ `ruleType(RULE(pat,exp),occ,loc)`
`=`

```
let val occ = Abstr occ
    val currentbound = (!rulebound)                (* added *)
    val (pat',pty) = patType(pat,lamdepth occ,loc)
    val _ = rbupdate pty                           (* added *)
    val (exp',ety) = expType(exp,occ,loc)
 in (rulebound := currentbound;                    (* added *)
     (RULE(pat',exp'),pty,pty --> ety))
 end
```

This is the base case for function expressions. In general a function consists of a list of rules of the form `RULE(pat,exp)`. Each is typed in turn by `ruleType`, and the results unified to give the type of the function.

In a call of `ruleType`, the current value of `rulebound` is recorded as `currentbound`. The pattern part of the rule is typed at a depth `lamdepth (Abstr(occ))` = `1 + lamdepth(occ)`, to give `pty`. `rbupdate` then adds the free variables of the pattern type to `rulebound` - see [17]. The expression part of the rule is then typed at `Abstr(occ)`, to give `ety`. The answer is then `pty --> ety`, and `rulebound` is reset to `currentbound`. Although, `rulebound` is reset to `currentbound`, `rulebound` can be side-effected by a call of `ruleType` – because some of the variables in `currentbound` may be side-effected.

31. `decType0(decl,occ,loc) : case decl of VALdec vbs =>`

```
    let fun vbType(vb as VB{pat, exp, tyvars=(tv as (ref tyvars))}) =
```

```
            let val (pat',pty) = patType(pat,infinity,loc)
                and (exp',ety) = expType(exp,occ,loc)
            in (unifyTy(pty,ety)
                handle Unify(mode) => ...);
                generalizePat(pat,tyvars,occ,loc);
                VB{pat=pat',exp=exp',tyvars=tv}
            end
     in VALdec(map vbType vbs)
    end
```

A non-recursive value declaration takes the form `VALdec vbs`, where `vbs` is a list of value bindings, each of the form `VB {pat,exp,tyvars }`. The `vbType` function defines the typing of an individual value binding, and the result for whole is obtained by mapping this function over the individual bindings. Note the result of typechecking a declaration is simply a refinement of the declaration.

Basically the pattern typing code assigns fresh type variables to the variable being declared, this is unified with the type of the definition, and the side-effected pattern type is then generalised. Note all calls share the same `occ` value with the outermost call.

The fresh type variables assigned to the pattern have depth *infinity*. See [21] for inheritance of depths under unification. Having depth infinity initially for variables in the pattern type ensures for all practical purposes that after the pattern type has been unified with the definition type, the generalisation of this pattern type will be exactly the same as the generalisation of the definition type[7]

32. ∼ `decType0(decl,occ,loc)  :  case decl of VALRECdec(rvbs) =>`

```
  let val thedeftys = ref([]:((ty ref * ty) list))
      fun setType(RVB{var=VALvar{typ,...}, resultty=NONE, ...}) =
     (typ := mkRefMETAty(1+ lamdepth occ))
          ...
      fun rvbType(rvb as RVB{var=v as VALvar{typ,...}, exp,resultty,tyvars}) =
             let val (exp',ety) = (expType(exp,Abstr(Rator occ),loc))
             in (thedeftys := ((typ,ety)::(!thedeftys)));
                 RVB{var=v,exp=exp',resultty=resultty,tyvars=tyvars}
             end
           ...
      fun genType(RVB{var,tyvars = ref tyvars,...}) =
             generalizeTy(var,tyvars,occ,loc)
      val doclean = if (!topprob) then (topprob:= false;true)
                           else false
      val _ = (app setType rvbs)
      val rvbs' = map rvbType rvbs
      val _ = let fun f (tref,ty) = unifyTy(!tref,ty)
               in (app f (!thedeftys))
              end
      val _ = (checkrvbpr (VALRECdec(rvbs)))
```

[7] The exception to this is if the declaration occurs embedded under more than 100000000 abstractions. It may well then arise that variables which should be generalised will acquire depth 10000000 through the unification step. They will then not be generalised, because the depth is less than the current depth.

```
    val _ = semiprob := (Semiunify.process(!semiprob)
                         handle Semiunify.SFAIL(mode,ineq,L) => ...)
    val _ = (app genType rvbs)
    val _ = if doclean  then (semiprob := [];
                              eqcounter := 1;
                              topprob := true) else ()
in VALRECdec rvbs'
end
```

A recursive value declaration takes the form `VALRECdec rvbs`, where `vbs` is a list of recursive value bindings, each of the form `RVB{f,fdef,resultty,tyvar list }`. There are following stages towards the final revised form of the declaration.

1. `setType` initialises the type-assumptions for the `f`'s. Unlike the non-recursive bindings, these initialisations are not with depth infinity variables, but with `1 + lamdepth occ`, cf. `ruleType`

2. `rvbType` then types the `def`'s in turn, each time updating the reference `thedeftys` with the pairing `(ftype,fdeftype)`. The definitions are typed by calling `expType` with an `occ` value `Abstr(Rator Root)`. This gives an `occ` value with `lamdepth` attribute incremented by 1, as well as altering other `weakness`-related fields.

3. After all the definitions have been typed, the `unifyTy` function is iterated over the `thedeftys` list, unifying each variables initialisation with its inferred type. Note this will automatically side-effect in the appropriate way the instantiation claims recorded in `semiprob`.

4. `checkrvbpr` is then called – see [8]. If `System.Control.CG.viewsemi` is set to true, the user will at this point be prompted with a requents whether the recursive value binding should be printed.

5. There is then a call to `Seminify.Process` – see [44] – on the current contents of `semiprob` - call it L. This will essentially specialise L, in a minimum way, to a true set of claims L' if that is possible, and then `semiprob` will be updated to have L' as value. `Semiunify.process` raises exceptions of the form `Semiunify.SFAIL(mode,ineq,L'')`, when no specialisation is possible. These exceptions are handled by printing an error message and then outputting L" as the result.

6. `genType` then generalises the set of inferred types, taking the initial value of `occ` as the parameter to the `generalizeTy` function. Thus generalisation is called at a *lower* `occ` value than that current during the typing of the definitions.

7. Finally a 'clean-up' clause is called, whose purpose is to detect whether a top-level declaration has just been typed. and if so to reinitialise certain values. Before a rec-binding is traversed, `topprob` is true. Within each call on a rec-binding, there is local variable `doclean`, which is set according to the global value of `!topprob`: if `!topprob` is true, `doclean is true`, if `!topprob` is false, `doclean` is false. As a side-effect of the case that `!topprob` is true, `topprob` is assigned to false. So on the embedded calls `!topprob` will be false, and therefore `doclean` will be false. Only if `doclean` is true (ie. top-level) is there a reinitialisation of the references `semiprob`, `eqcounter` and `topprob`.

33. \+ `The semiunifier code` :

We implements a version the Henglein/Leiss rewrite system defined in section 2. We alter the rewrites, so that where Henglein might add an identity $s = t$, instead apply $mgu(s, t)$ to the inequation system. The set of context variables is recorded in the reference `Semiunify.nonpat`, which is side-effected by the code in the `Typechecker` structure concerning `rulebound`.

See [45], [46], [47], [48], [57]

34. + `ineq` : `the datatype of inequalities: ineq of int*ty*ty`

    The int part is refered to as the index. Note `ineq(i,ty1,ty2)` will be pretty-printed as `ty1 <i ty2`, by `inpr` [4]

35. + `indexmatch` : `int list -> int -> bool`

    The empty list, and singleton integer lists are used as patterns for the indexes that inequalities have. The empty list matches any index. `[x]` matches `x`. Other cases raise the exception `Indexpat`

36. + `typematch` : `ty list -> ty -> bool`

    The empty list, and singleton type list are used as patterns for types. The empty list matches any index. `[x]` matches any `prune` equal `y`. Other cases raise the exception `Typepat`

37. + `matches` : `int list * ty list * ty list -> ineq -> bool`

    Component-wise matching of a pattern `(I,X,Y)` to an inequality

38. + `split` : `int list * ty list * ty list -> ineq list -> ineq list * ineq * ineq list`

    Given pattern `(I,X,Y)` and `S`, divides `S` into `(L,ineq(i,x,y),R)`, where `ineq(i,x,y)` is the first inequality matching `(I,X,Y)` in `S`; else raises the exception `SPLIT`

39. + `subtract` : `ineq -> ineq list -> ineq list`

    `subtract(ineq,L)` subtracts the first occurrence of `ineq` from `L`

40. + `find` : `int list * ty list * ty list -> ineq list -> ineq`

    given pattern `(I,X,Y)` and `S`, returns `ineq(i,x,y)` where `ineq(i,x,y)` is the first inequality matching `(I,X,Y)` in `S`; else raises the exception `FIND`

41. + `openvariable` : `ty -> bool`

    checks whether a `pruned` type is an open variable. If the type is of the form `VARty(ref(OPEN {kind=k,... }))`, then if k is user-bound, counted as not open, other wise open. If the type is of any other form, then it is not open.

42. + `show` : `(ineq list -> string -> ineq list) ref`

    This reference is initially assigned a trivial value. In the `Typecheck` module, then `show` is assigned to `checkpr`, which then optionally prints information given a semiunification problem. See [7] and [45]

43. + `thestring` : `string ref`

    Reference to a string which is updated after a rewrite step to a string describing the rewrite.

44. + `Process` :

    Called on a semiunification problem, initially applies `!show` to the problem and the string ``` ``initial problem'' ```, then applies `process` to the problem. The effect is that before rewriting commences, the user has an opportunity to see the initial problem. See [7].

45. + `process` :

This applies the rewrites `reduce1`, `reduce1a`, `reduce2` and `reduce3` exhaustively, with the priority `reduce1` > `reduce1a` > `reduce2` > `reduce3`. These corresponds to the rewrites S1-S4.

For each rewrite, when it cannot be applied it raises a particular exception, for which `process` has handling (roughly) as follows:

```
fun process ineqlist =
 process(reduce1(ineqlist,ineqlist)) handle Red1 =>
   process(reduce1a(ineqlist)) handle Red1a =>
    process(reduce2(ineqlist)) handle Red2 =>
     process(reduce3(ineqlist,ineqlist)) handle Red3 => ineqlist
```

note:

`Red1` handling only arises once `reduce1` does not apply
`Red1a` handling only arises once `(reduce1,reduce1a)` do not apply
`Red2` handling only arises once `(reduce1,reduce1a,reduce2)` do not apply
`Red3` handling only arises once `(reduce1,reduce1a,reduce2,reduce3)` do not apply

`reduce1` and `reduce3` have cases which raise the exception `SFAIL(mode,ineq,L)`, where `m` is a string recording what kind of a failure happened, `ineq` records the problematic inequation gave rise to the problem, and `L` is a certain semiunification system from which the problem inequation has been deleted. This is used to allow typechecker to continue after an error in order to detect further errors.

The `SFAIL(mode,ineq,L)` exceptions are not handled by `process`, but rather by the call to process from the typechecker – see [32].

Additionally, the `reduce` terms called by `process` are embedded as (`!show` *reduce term* `!thestring`). This evaluates to the `reduce` term, and according to the setting of `System.Control.CG.viewsemi`, allows the user to see the result after reduction. See [42],[7].

46. + `reduce1` : `ineq list*ineqlist -> ineq list`

uses `exception SFAIL(mode,ineq,ineqlist) exception Red1`

Essentially implements the S1 rewrite which replaces `(a1,...,an) f <i (b1,...,bn) f` in a semiunification problem with the inequatons `a1 <i b1,...,an <i bn`.

Initially `reduce1` is called by `process(L)`, and this sets the two arguments of `reduce1` equal. The output list is either immediately determined from the first argument, or by a recursive call `reduce1(tail(L),L)`. The second argument remains a record of the initial problem, and is used only in generating exceptions.

`reduce1(L1,L2)`:

Takes head `ineq(i,x,y)` of `L1`
tests whether `pruned` versions of `x` and `y` are functions
if *yes* tests whether functions are equal
if *yes* makes new inequations from the argument lists and appends to `(tl L1)`
if *no* raise `SFAIL("function clash in semiunify", ineq(i,x,y), subtract (ineq(i,x,y)) L2)`, so third argument of the exception is the initial problem minus the problematic inequality

if *no*, tests whether `x` is flexible record, and `y` is a `record` type.
if *yes* return `ineq(i,x',y)::  tl(L1)`, where `x'` is record-type derived from the flexible record `x` by adding the extra fields of `y`, typed with fresh-variables
if *no* return `ineq(i,x,y) ::  reduce1((tl L1),L2)`

If first argument is empty raise `Red1`. This has the effect that if `L` contains no possibility to apply the S1 rewrite and `process L` tries to rewrite `L` with `reduce1(L,L)`, the exception `Red1` will be raised.

47. + `reduce1a` : `ineq list -> ineq list`

uses `exception Red1a`

This basically implements the S2 rewrite, such that if `x <i y` occurs and x is a type-variable in the context set, then `x` should be unified with `y`. However, also variables arising from over-loaded symbols are also treated as if they were in the context set.

`reduce1a(L)` =

Takes head `ineq(i,x,y)` of `L`
tests whether if `x` is a zero-depth variable (and therefore types an occurrence of an overloaded symbol)
if *yes*, evaluates `unifyTy(x,y)`, returns `(tl L)`
if *no*, tests if `x` is in `!nonpat`,
if *yes*, evaluates `unifyTy(x,y)`, returns `(tl L)`
if *no* returns `ineq(i,x,y) ::  reduce1a(tl L)`

If `L` is empty, raises `Red1a`. This has the effect that if `L` contains no possibility to apply the S2 rewrite (and no LHS 0-depth variables) and `process L` tries to rewrite `L` with `reduce1a(L)`, the exception `Red1a` will be raised.

48. + `reduce2` : `ineq list -> ineq list`

uses `exception Red2`

Basically implements the S3 rewrite, rewriting a pair `x <i t` and `x <i s` to `x <i t`, with the side-effect that `t` and `s` are unified.

`reduce2(L)` =

takes head `(i,s,t)` of `L`
tests whether `s` is an open variable
if *yes*,use `([i],[s],[])` as pattern, `split` the rest of `L`, into `(Left,(i',s',t'),Right)` where `(i',s',t')` is first matching ineq, evaluate `unifyTy(t,t')`, return `(i,s,t)::(Left@Right)`
if *no*, return `(i,s,t)::  reduce2(tl L)`

When `L` is empty, `Red2` is raised. This has the effect that if `L` contains no possibility to apply the S3 rewrite, and `process L` tries to rewrite `L` with `reduce2(L)`, the exception `Red2a` will be raised.

49. + `makedtrs` : `ty * ineq list -> ty list`

An auxiliary function of `check` – see [52]. `makedtrs(y,L)` makes a list of all `x` such that `(x <_ y)` occurs on `L`. That is, the immediate daughters, or predecessors of `y` under `<` are collected.

50. + `minus` : `ty * ineq list -> ineq list`

An auxiliary function of `check` – see [52]. `minus(y,L)` discards inequalities `(y <_ _)` from `L`

51. + `iterate` : `('a -> bool) -> 'a list -> bool`

An auxiliary function of `check`, and `reduce3` – see [52], [57]. `iterate(f,L)` is true if `f x` is true for some `x` in `L`

52. + `check x y L` : `ty -> ty -> ineq list -> bool`

auxiliary function of `reduce3` – see [57] – which implements the 'extended occurs check' of the S4 rewrite: checks whether `x` can be linked to `y` by a chain of inequalities in `L`

`check x y S` =

> check whether `x = y`
> if *yes* then return `true`
> if *no* then `iterate` the function `fn dtr => (check x dtr S')` over `makedtrs(y,S)` (i.e. the immediate predecessors of `y` in `S`), where `S'` is `minus(y,S)` (i.e. `S` with inequalities `(y <_ _)` discarded)

Note, once we have a predecessor `z` of `y`, we wish to check whether `x` can be linked to `z`. In the search for such links we can ignore inequalities `(y <_ _)`, as linking `x` to `z` by means of such in inequality, means `x` can be linked to `y` more directly. This is the reason why in the definition of `check`, once linking to the predecessors of `y` is attempted, the pool of inequalities to be searched is reduced with `minus(y,S)`.

53. + `vars` : `ty list ref`

A reference counter used in `readrulebound` [15], `freevars` [54], and `reduce3` [57]

54. + `freevars` : `ty -> unit`

Auxiliary function of `readrulebound` [15] `reduce3` [57].

Given a type t, updates the type list ref, `vars`, adding all free variables t which are not already present in `vars`

55. + `newineqs` : `(ty*ty) list ref`

A reference value used in `reduce3`

56. + `Semiunify.copyTy` : `ty -> ty`

Auxiliary function `reduce3`, which copies a type by making fresh-copies of variables, updating also `newineqs` with the pairs consisting of the old and the new. User-bound variables are not copied. Only the typings of the known fields of a flexible record are copied.

57. + `reduce3` : `ineq list * ineq list -> ineqlist`

Implements the S4 rewrite, by eliminating `args f < x`, unifying `f(args')` with `x`, and adding inequalities `args < args'`, provided the 'extended occurs-check' succeeds

Initally `reduce3` is called by `process(L)`, and this sets the two arguments of `reduce3` equal. The output list is either immediately determined from the first argument, or by a recursive call `reduce3(tail(L),L)`. The second argument serves both as a pool of inequalities for the extended occurs-check, and is also in generating exceptions.

`reduce3(L1,L2)` =

takes head `ineq(i,t,v)` of `L1`
  checks whether `t = CONty(f,args)`, and `v` is an open variable
    if *yes* then checks whether `(check v y L2)` for free `y` of `t`
      if *yes* then raise `SFAIL("extended occurs check fail in semiunify", ineq(i,t,v), subtract (ineq(i,t,v)) L2)`
      if *no* then copy `t` to `t'`, evaluate `unifyTy(v,t')`, return `L'@ tl(L1)`, where `L'` are new inequations defined by the copying of `t` to `t'`
    if *no* then return `ineq(i,t,v)::reduce3(tl L1,L2)`

if `L1` is empty, raise `Red3`

5 Remarks on Design

5.1 Environments or variable attributes

As mentioned several times in the above, the SMLofNJ typechecker does without interim environments in typing bound variables. Instead types are inserted into the code, and the combination of the `depth` attribute of type variables and the `occ` argument of the typecheck functions is used to control the generalisation of type variables.

In `polyrec_sml`, we use what is effectively a reduced form of an interim environment - the `rulebound` list, indicating the type-variables present in currently in scope λ-bound variable. This is inelegant, as this rulebound list could be used to control type generalisation, and make the *depth* attribute of variables redundant.

A more elegant and more efficient design, might employ an alternative to the current datatype for types, in which type variables would have an additional attribute which would take over the work done by the `rulebound` list. Such a design was not implemented because of our overall decision not to redefine any of the datatypes of the compiler.

Attempts to exploit the existing depth attribute of type variables to implement the necessary control over type variable copying seemed doomed to failure. Consider

```
val rec f = fn x => let val y = f x in ....
```

The initial type variable assigned to `f` will have a depth attribute of 1, and that for `x`, a depth attribute of 2. However, after `f x` has been typed, the type variable for `x` will also have a depth attribute of 1. This means that as we descend further into the term, and perhaps encounter a variable tagged with a type featuring one of these variables, we cannot choose whether or not to copy on the basis of the depth attribute.

5.2 Overloading

The code below comes from util/feedback.

```
val infinity = 1000000000

fun minl l =
 let fun f(i,nil) = i | f(i,j::rest) = if i<j then f(i,rest) else f(j,rest)
  in f(infinity,l)
 end
```

Without taking special precautions it will cause the poly-rec typechecker an 'unresolved overload' problem.

Overloaded constants are treated in SMLofNJ as polymorphic constants. Whenever the constant occurs, unknowns of depth *zero* are chosen for the quantified variable, and global note is made of the unknown. Zero is a depth which can in no other way become associated with a type variable, and will never be generalised. In an embedded let, the definition type may contain unknowns of depth 0, and these may well not occur in the type of any wider scope abstracted variables. Such unknowns clearly should not be quantified, and this is achieved because of their zero depth.

Finally, at the top-level, after type-checking is effectively complete, there is final check whether the unknowns chosen at instances of overloaded constants have become instantiated. If they have not, the code is rejected as containing an unresolved overloading.

In the above code, the auxiliary function f will obtain type $X * Xlist \rightarrow X$, where X is a zero-depth unknown. In the abscence of polymorphic recursion, the subsequent use of f will cause the instantiation of X to int. Consider now the case for polymorphic recursion.

At the point where use of f is to be typed, in the usual way, the question arises as whether the unknowns in the assumed type for f should be copied or not. The basic criterion is that unknowns not occuring in the assumptions of currently in scope monomorphic assumptions should be copied. In the example, the unknown X does not occur in such a monomorphic assumption. However, if then the occurrence of f is typed via a copying, it is only the copy of X that will be instantiated to int, and not X itself. The result will be that at the final overload-check stage, X will remain uninstantiated. Clearly it is necessary to define the copying functions used by the type-checker (and by the semiunifier) so that the unknowns arising from occurrence of overloaded variables are treated as *monomorphic*: when such unknowns occur in a current typing assumption, they should *not* be copied in typing an occurrence. Also if such unknowns occur in a semiunification problem, they should be treated in the same way as the other monomorphic variables of the problem.

5.3 Flexible Records

Consider,

```
fun f = fn x => # 1 x
```

The definition of `f` can be assigned any type whose argument is a record with a field `1`, of some type τ, and whose value is τ. However, SML's type system cannot express the type quantification implicit in this description, and consequently the above declaration is rejected as not typable: the compiler's 'flexible record' types, serve only as temporary markers for a record type which must eventually be fully specified.

In `polyrec_sml` it remains the case that unresolved flexible records cannot be generalised. However, under polymorphic recursion, the process of resolution of flexible records can take a different course than it does under monomorphic recursion. Consider the following examples:

```
fun f x = (# 1 x; f(1,2); x)
fun f x = (# 1 x; f(1,2); f(‘a’,’b’); x)
```

Clearly, under monomorphic recursion the first receives type `int * int -> int * int`, whilst the second is untypable. This arises in the implementation because the type of the definition is unified with the types of the occurrences. Under polymorphic recursion, in both cases the function receives type `'a * b -> 'a * 'b`. For the first example, the input to `Semiunify.process` is:

```
{1:'A,...'Z }-> {1:'A,...'Z }<1 {1:int,2:int }-> 'B
```

Calling `Semiunify.process` on this, resolves the flexible record to be one with two fields, labelled `1` and `2`, but does not force these fields to have integer type. For the second example, the input to `Semiunify.process` is:

```
{1:'A,...'Z }-> {1:'A,...'Z }<1 {1:int,2:int }-> 'B
{1:'A,...'Z }-> {1:'A,...'Z }<2 {1:str,2:str }-> 'B
```

Again the flexible record is resolved to be one with two fields, labelled `1` and `2`, but types of these remain unconstrained.

6 Performance

We have tested `polyrec_ sml` by compiling all the sources for the compiler, both with and without polymorphic typechecking activated. The result is that the total compilation time is $\sim 10\%$ greater with polymorphic recursion than without. On the assumption that the source code of the compiler is representative of typical SML programming, this suggests that the average SML user could use `polyrec_ sml` without a punitive slowdown.

The file *doc/compiler_examples* notes some of the functions from the compiler sources that receive a different type under polymorphic recursion than they do under monomorphic recursion. There are ~ 30, the majority of which occur in simultaneous recursive declarations. A thorough analysis of the frequency of simultaneous recursive declarations would be useful here, but preliminary investigation of this on the basis of profiling the typechecker on the first $\sim 20\%$ of the compiler code, gives that less than 4 % of all recursively declared functions are part of a simultaneous recursion. This low frequency of simultaneous recursions may explain the relatively small number of cases in the compiler code which received a more general type under polymorphic recursion than under monomorphic.

Some of the examples from the source code resemble the first example in section 1.2. In these case the ‘real’ value of a recursive function is preceded by some side-effecting code which also invokes the recursive function. The side-effecting part constrains its instance more tightly than the definition part contrains its instance, giving a more restrictive type under monomorphic recursion than under polymorphic recursion.

This profiling also reveals that of the additional functions involved under polymorphic recursion, most time is spent in checking whether a given type variable is present in the `rulebound` list of variables. As mentioned in the previous section, altering the datatype for types might allow this check to be avoided. Also of the various possible rewrites on a semiunification problem, `Semiunify.reduce1` is called most often, then `Semiunify.reduce1a`, then `Semiunify.reduce2`, then `Semiunify.reduce3`.

The summaries after batch compiling with and without polymorphic recursion activated are given below.

```
60135 lines
parse      350.460000s
translate  34.610000s
codeopt    12.180000s
convert    13.650000s
cpsopt     237.870000s
closure    128.900000s
globalfix  9.300000s
spill      20.920000s
codegen    153.820000s
schedule   160.600000s
freemap    17.960000s
execution  0.0s
GC time    336.320000s
total(usr) 1472.570000s
total(sys) 27.070000s

60135 lines
parse      215.640000s
translate  35.830000s
codeopt    12.440000s
convert    13.980000s
cpsopt     243.890000s
closure    132.490000s
globalfix  9.470000s
spill      20.990000s
codegen    157.430000s
schedule   164.220000s
freemap    18.320000s
execution  0.0s
GC time    308.670000s
total(usr) 1329.530000s
total(sys) 22.270000s
```

References

[Altenkirch, 1995] Thorsten Altenkirch message to the ML-list, noting an example typable only with polymorphic recursion

[Damas and Milner, 1982] L.Damas and R.Milner. Principal type-schemes for functional programs. in *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212

[Elliot, 1991] Conal Elliot message to the ML-list, noting the `collect` example

[Emms and Leiss, forthcoming] M.Emms and H.Leiß Extending the SML Typechecker with Polymorphic Recursion: a correctness proof. CIS Technical Report

[Henglein, 1988] Fritz Henglein. Type inference and semiunification. in *Proceedings of the 1988 ACM Conf. on LISP and Functional Programming, Snowbird, Utah*, pp 184–197

[Kfoury, Tiuryn, and Urcyczyn, 1989] A.Kfoury, J.Tiuryn, P.Urcyczyn. *The Undecidability of the Semi-Unification Problem.* Boston University, Tech. Report BUCS-89-010

[Leiss, 1989] Hans Leiß Polymorphic Recursion and Semiunification. in *Proceedings of the 3rd Workshop on Computer Science Logic, CSL'89, Kaiserslautern FRG*, Springer LNCS 440, pp 211–224

[Milner, 1978] R.Milner. A Theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–378.

[Milner et al, 1990] R.Milner, M.Tofte and R.Harper *The Definition of Standard ML* MIT Press, Cambridge, Massachusetts

[Milner and Tofte, 1991] R.Milner and M.Tofte *Commentary on Standard ML* MIT Press, Cambridge, Massachusetts

[Mycroft, 1984] Alan Mycroft. Polymorphic Type Schemes and Recursive Definitions. in *Proceedings 6th International Conference on Programming, LNCS 167*