Universität München
Centrum für Informations- und
Sprachverarbeitung

Extending the Type Checker of SML by Polymorphic Recursion

A Correctness Proof

Martin Emms & Hans Leiß

CIS-Bericht-96-101

20. Januar 1997

Verantwortlich:
Petra Maier, Centrum für Informations- und Sprachverarbeitung,
Universität München, Oettingenstr. 67, 80538 München
EMAIL: pmaier@cis.uni-muenchen.de

Extending the type checker of SML by polymorphic recursion

A correctness proof

Martin Emms and Hans Leiß

{emms,leiss}@cis.uni-muenchen.de
Centrum für Informations-
und Sprachverarbeitung
Universität München
D-80538 Oettingenstr. 67, Germany

Abstract. We describe an extension of the type inference of Standard ML that covers polymorphic recursion. For any term t of *SML*, a type scheme τ and a system L of inequations between (simple) types is computed, such that the types of t are the instances of τ by substitutions S that satisfy L.
The inequation constraints L are computed bottom-up in a modification of Milner's algorithm W. The correctness proof is complicated by the fact that unknowns for polytypes are needed – in contrast to type inference for *SML*.

1 Introduction

Functional programming languages like *ML*[19], Miranda[23], or Haskell[?], have made statically typed polymorphic languages popular. Their success depends to a large extend on the following properties of the underlying type system of Damas/Milner[2]:

- typability of an untyped term is decidable,
- for typable terms, a schema representing the set of its types can be inferred automatically,
- the declaration of polymorphic values by the user is supported,
- well-typed terms do not cause run-time type errors.

However, these properties were achieved by restricting polymorphism somewhat: (i) *λ-abstraction is monomorphic:* a function may accept arguments of different (monomorphic) types, but the argument must be used with the same type at each occurrence in the body of the function, (ii) *Recursion is monomorphic:* at each occurrence in its definition, a recursive function x must be used with the (monomorphic) type τ of its defining term e:

$$\text{Rec Value Dec} \qquad \frac{\Gamma \cup \{x : \tau\} \vdash e : \tau}{\Gamma \vdash \texttt{val rec}\, x = e : \{x : \overline{\tau}^{\Gamma}\}},$$

where $\overline{\tau}^{\Gamma}$ is the universal closure of τ relative to the environment Γ and $\{x : \overline{\tau}^{\Gamma}\}$ the extension of the environment effected by the declaration.

While it is impossible to allow polymorphic λ-abstraction without loosing the existence of principal types, Mycroft[20] has shown that the Damas-Milner type system could be relaxed to allow polymorphic recursion. This Milner-Mycroft (or ML^+-) type system replaces the above rule of the Damas-Milner system by the rule

$$\text{(Poly)Rec Value Dec} \qquad \frac{\Gamma \cup \{x : \overline{\tau}^{\Gamma}\} \mid\!- e : \tau}{\Gamma \mid\!- \texttt{val rec}\ x = e : \{x : \overline{\tau}^{\Gamma}\}}.$$

The system still has the subject-reduction property, is type-sound, and has the principal types property. Mycroft proposed a semi-algorithm to compute principal types for ML^+-typable terms, but on every untypable term, this semi-algorithm diverges. Henglein[4, 5] and Leiß[13, 14] proposed semi-algorithms based on *semiunification*, which terminate also on all known examples of terms untypable in the Milner-Mycroft system. Actually, semiunification is equivalent to Milner-Mycroft-typability, cf. Henglein[5] and Kfoury e.a.[12].

But unfortunately, semiunification is an undecidable problem, as shown by Kfoury e.a.[11]. So the decidability of typability is lost under an extension of *ML* by polymorphic recursion. It therefore seems unreasonable to use the Milner-Mycroft type system in a real programming language: there are programs on which automatic type inference will not terminate.

However, an extension of *ML* by polymorphic recursion could still be of some practical value:

1. Some useful recursive programs can be typed in the Milner-Mycroft system, while they are untypable in the Damas-Milner system. Moreover, all terms typable in the Damas-Milner system are typable in the Milner-Mycroft system as well, and since the principal type is at least as general, the same program can be used in more situations when typed in the second system.
2. When polymorphic recursion –occasionally– is needed in practice, programmers seem to be irritated by the behaviour of *ML*'s type checker: not only is there an artifical distinction in polymorphism within and outside a recursive definition, but it is also impossible to make (current implementations of) *ML* respect a correct Milner-Myroft-type when provided by the programmer. (For examples of typing anaomalies raised by monomorphic recursion of *ML*, see Mycroft[20], Henglein[5], Kfoury e.a.[12].)
3. The importance of the complexity of type inference has to be judged from a practical point of view, not just from its theoretical worst-case analysis. As is well known, the type inference problem for *ML* is DEXPTIME complete (see [9, 16]), but the actual behaviour in programming practice for more than a decade led to the belief that it was linear. A similar difference between theoretical and practical complexity is expected to hold for type inference with polyporphic recursion (cf. Henglein[5]). The (limited) practical experience with our implementation of polymorphic recursion for *SML* showed no significant slow-down of type inference.

   Moreover, since program development is done interactively, the danger of a non-terminating compilation is low: a type checker consuming unreasonably much time or space resources would be stopped by the programmer anyway.

Adding polymorphic recursion to *SML* makes the type system more homogeneous: recursive functions are then type-uniform in the same way inside and outside of the definig term.

We think that only practice can tell which type discipline is most helpful in actual programming. Therefore, we have implemented type inference with the Milner-Mycroft system as a modification of the compiler of *SML* of New Jersey[1], a widely used version of *SML*. It allows the programmer to switch between the two type systems, and handles the full language of *SML* of New Jersey, including some features –such as flexible records and datatype declarations– that will not be discussed below. (For details, see the documentation Emms[3].)

The rest of this paper is organized as follows. In section 2, we will present definitions of the Damas-Milner and the Milner-Mycroft type systems, and recall its relation to semi-unification. Section 3 introduces a method of performing type inference for the Milner-Mycroft calculus in the style of algorithm $\mathcal{W}$ of Milner[18]. Section 3.2 gives an intuitive description of our semi-algorithm $\mathcal{W}^+$ for ML^+. Since it was intended for actual implementation, we also discuss some sublte points that arise from the aim of staying close to existing type checkers for *SML*. Section 3.3 gives the formal details of the algorithm. Its correctness and weak completeness are proven in section 4.

The methods for inferring Milner-Mycroft-types given by Henglein[5] and Kfoury e.a.[12] are in the style of Curry. They are discussed in the appendix, along with a variant of $\mathcal{W}^+$ and some implementation details.

2 The type systems of *ML* and ML^+

2.1 Terms and types

The term language we use is a λ-calculus with declarations, in the following notation of a sublanguage of *SML* (cf. Milner e.a.[19]):

Expressions	$e ::= x$		*variable*
	$\mid (e \cdot e)$		*application*
	$\mid$ (`fn` x `=>` e)		*function abstraction*
	$\mid$ (`let` d `in` e `end`)		*local declaration*
Declarations	$d ::=$ `val` x `=` e		*value declaratuion*
	$\mid$ `val rec` x `=` e		*recursive value declaration*

We refer to such terms also as *ML*-programs, ignoring some further restrictions met in *SML*, for example that the expression in a recursive value declaration has to be a function abstraction. Metavariables e, s, t are used to range over terms, and variables are taken from an enumerated sets $IVar$. (In the examples we also use appropiatly typed constants.)

As is done in *SML*, on the level of types we distinguish between

Monotypes	$\tau ::= \alpha$		*type variable*
	$\mid (\tau \to \tau)$		*function space*
Polytypes	$\overline{\tau} ::= \tau$		*unquantified type*
	$\mid \forall\alpha\overline{\tau}$		*quantified type*

We use $\alpha, \beta, \gamma, \delta$ for type variables, taken from an enumerated infinite set TV, and ρ, σ, τ for mono- and $\overline{\sigma}, \overline{\tau}$ for polytypes. By $FV(\overline{\tau})$ we mean the set of *type variables free in* $\overline{\tau}$.

An *environment* Γ is a finite partial function from individual variables to polytypes, written as a set of *typing statements* $x : \overline{\tau}$. The *type variables free in* Γ are

$$FV(\Gamma) := \{\alpha \mid \text{for some } x : \overline{\tau} \text{ in } \Gamma,\ \alpha \in FV(\overline{\tau})\}.$$

A type *substitution* S is a partial function from type variables to monotypes. A monotype τ *is a (generic) instance of* $\overline{\sigma}$, written as $\overline{\sigma} \succ \tau$, if τ results from $\overline{\sigma}$ by instantiating all bound variables of $\overline{\sigma}$ with monomorphic types, i.e. if $\tau = S\sigma$ for the quantifier free part σ of $\overline{\sigma}$ and a substitution S which is the identity on $FV(\overline{\sigma})$. We say $\overline{\sigma}_1 \succ \overline{\sigma}_2$, if $\overline{\sigma}_1 \succ \tau$ for each monotype τ such that $\overline{\sigma}_2 \succ \tau$.

For an environment Γ and a polytype $\overline{\tau}$, by $\overline{\tau}^{\Gamma}$ we mean the *universal closure of* $\overline{\tau}$ *with respect to* Γ, obtained from $\overline{\tau}$ by quantifying (in the order of occurrence from left to right) all its free type variables except those of $FV(\Gamma)$.

CONVENTION: The bound individual variables of a term are pairwise distinct, and disjoint from its free variables. In $S\overline{\tau}$, we assume that S is the identity on the bound variables of $\overline{\tau}$, and that none of these is a free variable of the image of S. Hence, if $\overline{\tau} = \forall \boldsymbol{\beta}\, \tau$, then $S\overline{\tau} = \forall \boldsymbol{\beta}\, S\tau$.

2.2 The Damas-Milner and Milner-Mycroft type systems

The set $\{\, \tau \mid \Gamma \vdash e : \tau \,\}$ of *types of a term* e, relative to an environment Γ that contains an assumption $x : \Gamma(x)$ for each x free in e, is inductively defined by a type-assignment calculus, see Figure 1. The first six typing rules define[1] the *Damas-Milner-calculus*, which we will also refer to as *ML*. The *Milner-Mycroft-calculus*, referred to as ML^+, is obtained by replacing the sixth rule by the last one.

Lemma 1. *In ML and* ML^+ *we have for any substitution* S*:*
If $\Gamma \vdash e : \tau$ *then* $S\Gamma \vdash e : S\tau$*. If* $\Gamma \vdash d : \{x : \overline{\tau}^{\Gamma}\}$ *then* $S\Gamma \vdash d : \{x : S(\overline{\tau}^{\Gamma})\}$.

Proof By induction on e resp d. Consider a polymorphic recursive declaration and assume $\Gamma \vdash \texttt{val rec}\ x = e : \{x : \overline{\tau}^{\Gamma}\}$. By induction, $S\Gamma \cup \{x : S(\overline{\tau}^{\Gamma})\} \vdash e : S\tau$. If $\overline{S\tau}^{S\Gamma} = S(\overline{\tau}^{\Gamma})$, the claim follows by an application of the typing rule. But since bound variables do not occur in the domain or range of substitutions, $\overline{S\tau}^{S\Gamma} = S(\overline{\tau}^{\Gamma})$ holds. □

Lemma 2. *Suppose* $\overline{\sigma}_1 \succ \overline{\sigma}_2$*. If* $\Gamma, y : \overline{\sigma}_2 \vdash e : \tau$*, then* $\Gamma, y : \overline{\sigma}_1 \vdash e : \tau$*.*
If $\Gamma, y : \overline{\sigma}_2 \vdash d : \{x : \overline{\tau}^{\Gamma, y:\overline{\sigma}_2}\}$*, then* $\Gamma, y : \overline{\sigma}_1 \vdash d : \{x : \overline{\tau}^{\Gamma, y:\overline{\sigma}_1}\}$.

[1] The calculi of Milner[2] and Mycroft[20] use ∀-introduction and -elimination rules on the right of ⊢. We work with syntax-directed versions like the one of Milner e.a.[19], for the sublanguage we are concerned with. Note that the quantified variables in inferred typing statements of declarations are fixed.

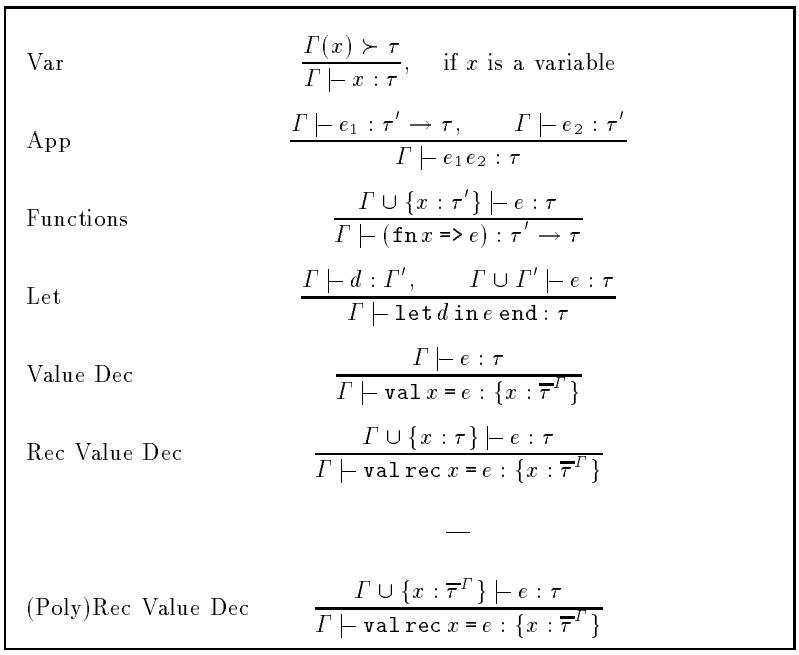

Fig. 1. Typing rules for *ML* and ML^+

Proof Again, we only consider a polymorphic recursive declaration. Suppose

$$\Gamma, y : \overline{\sigma}_2 \vdash \mathtt{val\,rec}\, x \,\mathtt{=}\, e : \{x : \overline{\tau}^{\Gamma, y:\overline{\sigma}_2}\}.$$

By the typing rule, me must have $\Gamma, y : \overline{\sigma}_2, x : \overline{\tau}^{\Gamma, y:\overline{\sigma}_2} \vdash e : \tau$. As $\overline{\sigma}_1 \succ \overline{\sigma}_2$, we have $FV(\overline{\sigma}_1) \subseteq FV(\overline{\sigma}_2)$, hence $\overline{\tau}^{\Gamma, y:\overline{\sigma}_1} \succ \overline{\tau}^{\Gamma, y:\overline{\sigma}_2}$. Therefore, by induction we get $\Gamma, y : \overline{\sigma}_1, x : \overline{\tau}^{\Gamma, y:\overline{\sigma}_1} \vdash e : \tau$. By the typing rule, this implies the claim. □

2.3 Examples of polymorphic recursion

Two defects of monomorphic recursion that are overcome by polymorphic recursion are demonstrated by the following examples. The first shows that while *SML* allows the definition of datatypes with increasing type parameters, it forbids recursion along the structure of its data. The second shows that monomorphic recursion reduces the power of simultaneous definitions.

Example 1. Recursive functions on recursively defined datatypes whose type parameters increase at the recursive occurrence:

Suppose[2] we store data of type α, indexed by structured keys, in a datastructure α-*trie*:

```
datatype key =
     Atom of int | Pair of key * key
datatype 'a trie =
     Empty | Branch of ((int * 'a) list) * (('a trie) trie)
```

For atomic keys, the data are stored in the first component, a list, while for pairs of keys, the data for all keys with the same first component are grouped into a trie organized according to the second component of keys. The (partial) function

```
fun find (Branch((c,a)::l,_), Atom(d)) =
         if d=c then a else find (Branch(l, Empty), Atom(d))
  | find (Branch(_,t), Pair(p,q)) = find (find (t,p), q)
```

is untypable in *SML*: since α-*trie*s contain (α-*trie*)-subtries, functions recurring along the structure of α-*trie*s need more complex types at recursive calls than at top level. The principal Milner-Mycroft type of *find* is $\alpha\text{-}trie \times key \rightarrow \alpha$, as one would expect.

Example 2. In simultaneous recursive definitions, a residual function sometimes needs different types at different occurrences. In the following example, f is needed with the types $\alpha\text{-}list \rightarrow \alpha\text{-}list$ and $\beta\text{-}list \rightarrow \beta\text{-}list$:

```
fun F (x : ''a list, atob : ''a -> ''b, bstoa : ''b list -> ''a) =
    let val rec f = fn x => if x = [] then []
                            else (bstoa (g x))::(f (tl x))
        and     g = fn x => if x = [] then []
                            else (atob (hd x))::(f (g (tl x)))
    in f x end;
```

In *ML*, F can be typed only when $\alpha = \beta$ (where $''\alpha$ means that there is a decidable equality on type α), with principal type

$$\vdash_{ML} F : \alpha\text{-}list \times (\alpha \rightarrow \alpha) \times (\alpha\text{-}list \rightarrow \alpha) \rightarrow \alpha\text{-}list,$$

imposing unnecessary restrictions on the use of F. In ML^+, we have the more general typing

$$\vdash_{ML^+} F : \alpha\text{-}list \times (\alpha \rightarrow \beta) \times (\beta\text{-}list \rightarrow \alpha) \rightarrow \alpha\text{-}list.$$

Mycroft[20] reports an example of an *ML*-untypable simultaneous recursion that arose from trying to avoid duplicated code. Also, *ML* conflicts with using simultaneous definitions to structure a program: the simultaneous definition

[2] We simplify an example reported by C.Elliot on the SML-electronic forum in 1991, who stores information indexed by terms and adds continuation functions to cover failure cases. R.Milner and P.Wadsworth have met similar examples in the development of the first ML-implementations in the seventies.

```
fun f(x,y) = (g(0,x),g(true,y)) and g(u,v) = v
```

is untypable in ML, since the residual function g is used with inconsistent types within the definition. While

$$\vdash_{ML^+} \quad f : \alpha \times \beta \to \alpha \times \beta, \qquad g : \alpha \times \beta \to \beta,$$

one can type f under the monomorphic recursion of ML only when defining g independently.

2.4 Polymorphic recursion and semiunification

Where type inference for ML depends on solving equational constraints of types, type inference for ML^+ additionally involves solving matchability constraints. In a ML^+-derivation

$$\frac{\begin{array}{c}\vdots \\ \Gamma,\ x : \overline{\tau}^{\Gamma} \mid\!- e(\ldots x^{\tau_1} \ldots x^{\tau_n} \ldots) : \tau\end{array}}{\Gamma \mid\!- \texttt{val rec}\ x = e : \{x : \overline{\tau}^{\Gamma}\}}, \tag{1}$$

the types τ_i of the occurrences of x in e are generic instances of the assumed type $x : \overline{\tau}^{\Gamma}$, i.e.

$$\overline{\tau}^{\Gamma} \succ \tau_1, \ldots, \overline{\tau}^{\Gamma} \succ \tau_n.$$

Since the derived type τ of e is the quantifier free part of $\overline{\tau}^{\Gamma}$, this means there are substitutions $T_1, \ldots, T_n$ with

$$T_i\tau = \tau_i \quad \text{and} \quad T_i\beta = \beta \qquad \text{for } \beta \in FV(\overline{\tau}^{\Gamma}),\ i = 1, \ldots, n. \tag{2}$$

Definition3. A substitution T *satisfies* the *matching statement* $\tau \sqsubseteq \sigma$ between monotypes τ, σ, if $T\tau = \sigma$. The restriction of T to $FV(\tau)$ will be called *the matching substitution* of $\tau \sqsubseteq \sigma$.

Hence

$$\overline{\tau}^{\Gamma} \succ \tau_i \iff \text{some } T_i \text{ satisfies } \{\tau \sqsubseteq_i \tau_i\} \cup \{\, \gamma \sqsubseteq_i \gamma \mid \gamma \in FV(\Gamma)\,\}. \tag{3}$$

In order to *infer* ML^+-types for recursive declarations $\texttt{val rec}\ x = e$, an environment $\Gamma = U\Delta$ and suitable monotypes $\tau = U\alpha, \tau_1 = U\alpha_1, \ldots, \tau_n = U\alpha_n$ will be obtained from the solution U of a set of constraints. These constraints are partly determined by e and additionally contain

$$\alpha \sqsubseteq_1 \alpha_1, \ldots, \alpha \sqsubseteq_n \alpha_n, \qquad \delta \sqsubseteq_1 \delta, \ldots, \delta \sqsubseteq_n \delta, \quad \delta \in FV(\Delta)$$

with type unknowns $\alpha, \alpha_1, \ldots, \alpha_n$ and an initial environment Δ. These matchability constraints ensure that the solution satisfies the condition (2) (where $\beta \in FV(U\delta)$) forced by the typing rule.

Definition 4. A *semiunification problem* in $\sqsubseteq_1, \ldots, \sqsubseteq_n$ is a multi-set L of equations, $\tau = \sigma$, and inequations, $\tau \sqsubseteq_i \sigma$ with $1 \leq i \leq n$, between monotypes τ and σ. A *solution* of L is an $n+1$ tuple $(U, T_1, \ldots, T_n)$ of substitutions, such that

$$U\tau = U\sigma \text{ for each } \tau = \sigma \text{ in } L, \text{ and } T_i U\tau = U\sigma \text{ for each } \tau \sqsubseteq_i \sigma \text{ in } L,\ i = 1, \ldots, n.$$

Call U a *semiunifier of* L and $T_1, \ldots, T_n$ its *residual matching substitutions.*[3] The solution $(U, T_1, \ldots, T_n)$ is *more general than* a solution $(U', T_1', \ldots, T_n')$, if $U' =_{FV(L)} RU$ for some substitution R.

We say L *holds*, if Id is the semiunifier of some solution $(Id, T_1, \ldots, T_n)$ of L.

Remark 1. Note that semiunification does not behave like unification:

(a) If L holds and S is some substitution, SL may not hold, it may even be unsolvable: take $L = \{\alpha \sqsubseteq \beta \to \beta\}$ and $S = [(\gamma \to \delta) \to \gamma/\alpha]$.

(b) If S is a most general semiunifer of L, then SL may have more variables than L: take $L = \{\alpha_1 \to \alpha_2 \sqsubseteq \beta\}$ and $S = [\beta_1 \to \beta_2/\beta]$. (The semiunifier $S' = [\alpha_1 \to \alpha_2/\beta]$ is not most general.)

By the above, it is clear that semiunifiability and ML^+-typability are closely related. In fact, for a variant of ML^+ without declarations and recursive *expressions* `rec` x `=` e instead –called ML^+ for the rest of this section–, Henglein[5] and Kfoury e.a.[12] have shown:

Theorem 5. *Henglein[5], Kfoury e.a.[12] The problems*

$$ML^+P := \{\, e \mid \Gamma \vdash_{ML^+} e : \tau \text{ for some } \Gamma, \tau \,\} \quad \text{and}$$
$$SUP := \{\, L \mid \text{some } (U, T_1, \ldots, T_n) \text{ solves } L \,\}$$

are polynomial-time reducible to each other.

The reduction of ML^+P to SUP uses a type inference method in the style of Curry: to each subterm t of e, one associates a type variable α_t and a finite set L_t of equational or matchability constraints. From a semiunifier U of L_e one gets $\Gamma = \{\, x : U(\alpha_x) \mid x \text{ is free in } e \,\}$ and $\tau = U(\alpha_e)$ such that $\Gamma \vdash_{ML^+} e : \tau$.

However, one has to be careful in defining L_t, since these constraints depend not only on the subexpression t and its position in e: if t is an occurrence of a `rec`-bound variable x, *the environment of the subexpression* `rec` x `=` s *binding* x has to be taken into account (cf. Γ occurs in (2). A similar remark applies to `let`-bound variables x.).

Remark. This has been overlooked in Henglein[5], whence his algorithm needs a correction (see Section 5.1). The constraints given by Kfoury e.a.[12] take the scope relations of bindings into account. However, this proof is via an embedding of ML^+ into λ-calculus with

[3] Note that T_i is a *simultaneous* matching for all $U\tau \sqsubseteq_i U\sigma$ with $\tau \sqsubseteq_i \sigma$ in L.

polymorphic recursion *and* polymorphic abstraction, but no `let`, which is then reduced to SUP. Therefore, it gives the constraints needed for ML^+-type inference only implicitly.

The interplay of monomorphic λ-binding and polymorphic `rec`-binding as well as the handling of declarations in the scope of recursions deserve a more explicit treatment. This, and the connection to principal ML^+-types, is given in our semi-algorithm $\mathcal{W}^+$ below.

For the sake of completeness, we recall some known properties of semiunification:

Theorem 6. *1. Each solvable instance of the semiunification problem has a most general solution.(Pudlák[21])*

2. *There are rewriting procedures that turn any solvable instance of the semiunification problem into a 'solved form', from which a most general solution can be extracted. (Henglein[5],Leiß[14],Kfoury e.a.[?])*
3. *The semiunification problem is undecidable for instances with at least two inequation relations and at least one binary function symbol. (Kfoury e.a.[11])*
4. *The semiunification problem is decidable*
   a) for instances with only one inequation relation. (Kapur e.a.[8]),
   b) for instances with only monadic function symbols. (Henglein/Leiß[15])

We remark that 4 a) implies that Milner-Mycroft-typability is decidable for *linear* polymorphic recursion, i.e. when a recursive function occurs only once in its defining term. A similar subclass cannot be derived from 4 b), since the binary $\rightarrow$ is the main constructor of type expressions.

We will only use most general semiunifiers satisfying a slight restriction. A semiunifier U of L *trivially renames* α, if $U\alpha \neq \alpha$ is a variable and $U\alpha \notin FV(U\beta)$ for all variables $\beta \neq \alpha$ in L. For example, if $L = \{\gamma \rightarrow \gamma \sqsubseteq \alpha \rightarrow \beta\}$ and $\alpha, \beta, \gamma, \delta$ are different, $U = [\beta/\alpha, \beta/\beta, \delta/\gamma]$ trivially renames γ, but not α.

Proposition 7. *If L has a solution, then it has a most general semiunifier that does not trivially rename variables of L.*

Proof By induction on the number of variables of L that are trivially renamed by a most general semiunifier. Suppose $(U, T_1, \ldots, T_n)$ is a most general solution of L where U trivially renames α to $\alpha' = U\alpha$. Let $\tilde{\alpha}$ be a fresh variable, and

$$\tilde{U} := [\tilde{\alpha}/\alpha, \alpha/\alpha']\, U, \qquad \tilde{T}_i := [\tilde{\alpha}/\alpha, \alpha/\alpha']\, T_i\, [\alpha/\tilde{\alpha}, \alpha'/\alpha].$$

Then $(\tilde{U}, \tilde{T}_1, \ldots, \tilde{T}_n)$ is a solution of L. It is a most general solution, since if a semiunifier U' of L equals RU on L, it equals $(R[\alpha/\tilde{\alpha}, \alpha'/\alpha])\tilde{U}$ on L. Suppose γ in L is trivially renamed by $\tilde{U}$, but not by U. Then $U\gamma$ must be a variable, and either $\gamma = U\gamma$ or $U\gamma \in FV(U\beta)$ for some $\beta \neq \gamma$ in L. In the second case, it follows that $\tilde{U}\gamma \in FV(\tilde{U}\beta)$, so $\tilde{U}$ would not trivially rename γ. In the first case, we have $\gamma \notin \{\alpha, \alpha'\}$ since U renames α to α'; but since $\tilde{U}\gamma \neq \gamma$, we must have $\gamma = U\gamma \in \{\alpha, \alpha'\}$, a contradiction. Since $\tilde{U}\alpha = \alpha$, $\tilde{U}$ trivally renames less variables of L than U. □

Remark. To type *ML*'s `let` $x = e$ `in` s `end`, one also has to find a polytype $\overline{\tau}$ for x together with instances $\overline{\tau} \succ \tau_1, \ldots, \overline{\tau} \succ \tau_n$ for occurrences of x in s. This can also be done in the style of Curry, i.e. using equational and matchability constraints, see Kfoury e.a.[10]. The constraints that arise this way are so-called *acyclic* semiunification problems, a decidable subproblem of semiunification.

3 Milner-style type inference of polymorphic recursion

As noted above, type inference for ML^+ can in principle be done in the style of Curry. In order to extend compilers for *ML* by polymorphic recursion, however, one has to stay closer to existing type checkers of *ML*. These are based on Milner's[18] type inference algorithm $\mathcal{W}$, which differs from Curry's style in some respects.

3.1 Milner's type inference algorithm $\mathcal{W}$ for *ML*

Milner's $\mathcal{W}$, which we split into $(\mathcal{W}_{dec}, \mathcal{W}_{exp})$ to fit to the distinction between declarations and expressions in *SML*, takes an environment Γ and an expression e (resp. a declaration d) and returns a failure report or a pair (S, τ), consisting of a substitution S to update the environment Γ and a type τ for e (resp. a pair $(S, \{x : \overline{\tau}\})$ with a typing statement $x : \overline{\tau}$ for d) (see Figure 2). (S can be seen as a solved form of the equational constraints in a Curry-style method.)

An algorithm *mgu* that computes a *most general unifier* of two types is used to construct S. (Cf. Herbrand[6], Robinson[22], Martelli[17].)

For comparison with our extension $\mathcal{W}^+$ for ML^+, we state the well-known properties of $\mathcal{W}$. We say $S =_\Gamma T$ if the substitutions S and T agree on $FV(\Gamma)$. $S\Gamma$ is obtained from Γ by replacing assumptions $x : \overline{\tau}$ by $x : S\overline{\tau}$. A *typing of expression e modulo Γ* is (represented by) a pair (S, τ) such that $S\Gamma \vdash_{ML} e : \tau$. Similarly for declarations.

Theorem 8. *(Damas/Milner[2]) Let Γ be an environment that contains an assumption for each free variable of the expression e resp. declaration d. Then*

(a) If $\mathcal{W}_{exp}(\Gamma, e) = (S, \tau)$, then for all (S', τ') the following are equivalent:

(i) $S'\Gamma \vdash_{ML} e : \tau'$,

(ii) For some substitution U, $S' =_\Gamma US$ and $\tau' = U\tau$.

(b) If $\mathcal{W}_{dec}(\Gamma, d) = (S, \{x : \overline{\tau}\})$ and $\overline{\tau} = \forall\beta\tau$, then for each $(S', \{x : \overline{\sigma}\})$ the following are equivalent:

(i) $S'\Gamma \vdash_{ML} d : \{x : \overline{\sigma}\}$,

(ii) For some substitution U, $S' =_\Gamma US$ and $\overline{\sigma} = \overline{U\tau}^{USF}$.

(c) If $\mathcal{W}_{exp}(\Gamma, e) = fail$, there is no (S', τ') such that $S'\Gamma \vdash_{ML} e : \tau'$. If $\mathcal{W}_{dec}(\Gamma, d) = fail$, there is no $(S', x : \overline{\sigma})$ such that $S'\Gamma \vdash_{ML} d : \{x : \overline{\sigma}\}$.

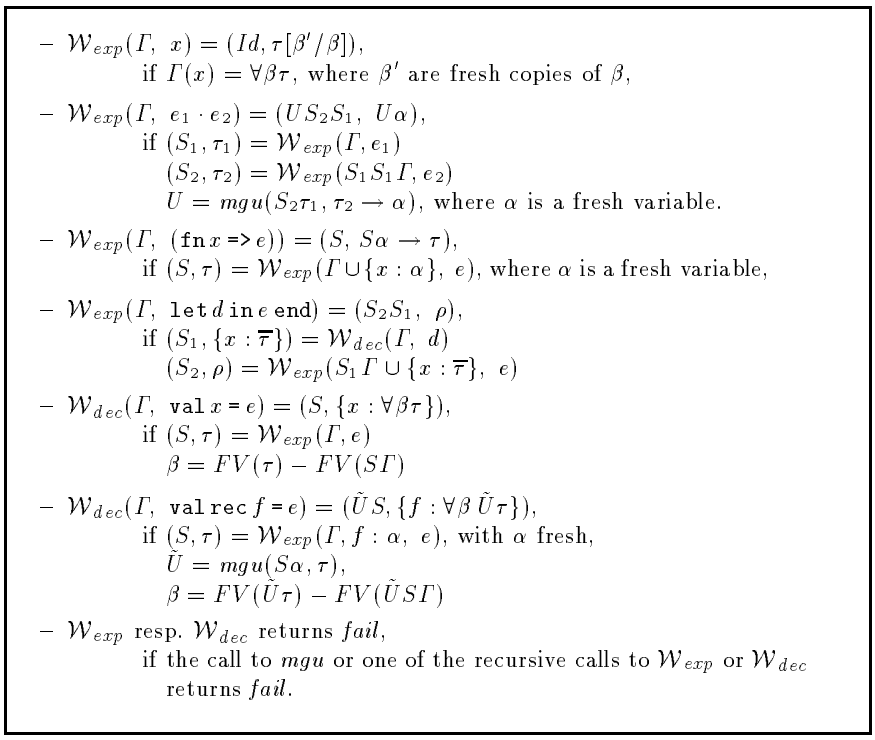

Fig. 2. The type assignment function $\mathcal{W}$ for *ML*

From $\mathcal{W}_{exp}(\Gamma, e) = (S, \tau)$ one obtains $S\Gamma \vdash_{ML} e : \tau$, the *principal ML-type of* e *modulo* Γ, by choosing $U = Id$ in (ii). The corresponding holds for declarations.

In the proof, one shows that the substitutions combined during the recursive calls of $\mathcal{W}$ correspond to combinations of and substitutions into *ML*-derivations.[4] These arise from solving equational type constraints (for application expressions and recursive declarations). The proof uses that if U is a unifier for an equation, so is any refinement SU.

For $\mathcal{W}^+$ and ML^+-derivations below, the situation is more complicated: when typing a recursively declared variable x, we have to weaken a temporary assumption $x : \overline{\tau}$ to some suitable $x : \overline{\sigma}$ with $\overline{\tau} \succ \overline{\sigma}$. Therefore, Lemma 1 will not be sufficient to prove our analog of Theorem 8 for ML^+.

Remark. Observe that $\mathcal{W}$ can type `let`-expressions without setting up inequation constraints, because it traverses its input terms in a specific order: for `let` d `in` e `end`, the declaration d is typed before its scope e is, introducing a polytype assumption $x : \overline{\tau}$ in the environment for e. All type variables free in $\overline{\tau}$ are *mono*type parameters of the derivation, since they occur as well in the type of some (monomorphic) λ- or `rec`-bound variable with

[4] Lemma 1 for *ML* is implicitly contained in the direction (ii) $\Rightarrow$ (i) of Theorem 8.

wider scope than x. The quantifier structure of these assumptions $x : \overline{\tau}$ need not to be changed when derivations are instantiated.

3.2 An informal description of $\mathcal{W}^+$ for ML^+

Our aim is to provide an extension $\mathcal{W}^+$ of Milner's $\mathcal{W}$ that allows one to infer principal ML^+-types for actual programs written in *SML*, if they are ML^+-typable, and to detect their ML^+-untypability as often as possible, otherwise; whether examples of nontermination –predicted by theory, but as yet unconstructed– also occur in practice, remains to be seen. Staying close to $\mathcal{W}$ has several advantages over an algorithm in the style of Curry/Hindley:

- an implementation is easier to realize for a full programming language, by modifying the code of an existing typechecker for *SML* (as we did for *SML of New Jersey*),
- the behaviour of the typechecker should not differ from the one for *SML* for recursion-free programs; in particular, `let`-expressions would not envolve the solution of (acyclic) semiunification problems,[5]
- the sequential nature of the typechecker should make error reports more specific, since it would solve inequation constraints at each recursive declaration rather than at the top node of a term.

A sequential treatment of type inference for ML^+, however, raises two problems that have to be explained to make the code of $\mathcal{W}^+$ below understandable.

Problem 1 To infer a principal ML^+-typing for a recursive declaration, `val rec` x `=` e, modulo an environment Γ, one has to introduce a *polytype unknown* α in order to first infer a principal type τ of the defining expression e modulo the environment $\Gamma, x : \forall\alpha.\alpha$. Knowing τ, the strong assumption $x : \forall\alpha.\alpha$ will be weakened to a suitable $x : \overline{\sigma}$ (with $e : \overline{\sigma}$). Hence:

1. The specialization of derivations needed is not just the substitution of free type parameters by monotypes, but also the weakening $x : \overline{\tau} \succ x : \overline{\sigma}$ of polytype assumptions, which we describe by substitutions instantiating polytype unknowns. For any inferred typing, $\mathcal{W}^+$ outputs a constraint set L that defines 'specializing' substitutions: only semiunifiers of L turn the inferred typing into a weakened ML^+-typing.
2. For declarations d in e, the quantifiers of their inferred typings $d : \{y : \forall\beta.\sigma\}$ can *not* be obtained –as in *ML*– by quantifying all variables not in the environment Γ at d: type variables of σ that occur in instances of the assumption $x : \forall\alpha.\alpha$ have to be treated as if they occurred in Γ, in order to respect the possiblity of later weakenings $x : \forall\alpha.\alpha \succ x : \overline{\tau}$.

   Therefore, the clauses of $\mathcal{W}^+_{dec}$ and $\mathcal{W}_{dec}$ differ in the type quantifiers they introduce: in $\mathcal{W}^+_{dec}$, only those variables of the type of the defining expression are quantified that are known to never occur in specializations of the current environment.

[5] The clause for `let`-expressions in $\mathcal{W}^+$ essentially is the same as in $\mathcal{W}$. Still, the derived types for locally declared variables may differ, because a `let`-expression embedded in recursive declarations may contain a polymorphic recursion variable - whose poly(!)type is only approximately known when typing the `let`-expression.

Note also that if d is `val(rec)` $y = t$ and x occurs in t, then any occurrence of y in the scope of d contains an implicit occurrence of x in e, which has to enter the typing of x.

Problem 2 In a Curry-style type inference for ML^+, the typing of an occurrence of a `rec`- or `let`-bound variable x involves matchability constraints about types in the environment of the declaration resp. expression binding x. To stay close to implementations of $\mathcal{W}$, we avoid introducing scope relations in the environment (cf. section 5.1) or attributing variables with the environment of their binding position. Instead, we keep the typing of variables simple and shift the burden to the (less frequent) typing of λ-expressions:

1. The input environment is split into two partial functions, one for λ-bound variables, and the other for `rec/let`-bound variables (possibly containing assumptions with unknown polytypes).
2. When typing an occurrence of a `rec`- or `let`-bound x, we add matchability constraints for all types of λ-assumptions in the environment *of the occurrence of* x, not just for those in the environment of the binding of x.
3. When typing a λ-subexpression e of the declaration resp. expression of a `rec`- or `let`-bound variable x, we retract the matchability constraints for types of the λ-assumption that were added when typing occurrences of x in e.

Thus, typing proceedes as if all λ-bound variables of an environment had wider scope than all its `rec`- and `let`-bound variables. When discharching a λ-assumption, part of the true scope relations becomes available, and matchability constraints are adjusted. Therefore, in $\mathcal{W}_{exp}$ and $\mathcal{W}^+_{exp}$the clauses both for variables and for λ-expressions differ.

Some of the above aspects will become clearer from an informal description of the typing of a recursive declaration.[6] The typing of `val rec` x `=` e is done in two phases:

Phase 1: From $\Gamma, x : \alpha$ and e, $\mathcal{W}^+_{exp}$ infers a type τ, a substitution S with $S\alpha = \alpha$, *and* a set L of inequations. The result (L, S, τ) represents a derivation which ends in

$$\langle S, \Gamma \cup \{x : \alpha\}\rangle = \ \langle S, \Gamma\rangle \cup \{x : \forall\alpha.\alpha\} \ \vdash_{ML^+} e : \tau, \tag{4}$$

under an appropriate specialization $\langle S, \Gamma \cup \{x : \alpha\}\rangle$ of $\Gamma \cup \{x : \alpha\}$, with $\forall\alpha.\alpha = \overline{S\alpha}^{\langle S,\Gamma\rangle}$. Its leaves are represented by L, recording matchability constraints to be satisfied under any weakening $\langle TS, \Gamma \cup \{x : \alpha\}\rangle$ of $\langle S, \Gamma \cup \{x : \alpha\}\rangle$. (The assumed polytype unknowns never occur in derived type expressions, but they will occur in the constraints L.)

Phase 2: To fit the typing rule, the assumption $x : \forall\alpha.\alpha$ in (4) has to be weakened, so that the quantifier-free part of the polytype of x is an instance of the derived type τ of e. We solve the constraints $L[\tau/\alpha]$ by a most general semiunifier U, if possible. Then $(U(L[\tau/\alpha]), US, U\tau)$ represents a weakened ML^+-derivation ending in

$$\langle US, \Gamma \cup \{x : \tau\}\rangle \ = \langle US, \Gamma\rangle \cup \{x : \overline{U\tau}^{\langle US,\Gamma\rangle}\} \vdash_{ML^+} e : U\tau, \tag{5}$$

[6] We ignore the splitting of the environment, which is relevant also in defining the updated environment, and hence write $\langle S, \Gamma\rangle$ for the updated environment. Details are adjusted later.

in which the instantiation conditions for typing occurrences of x,

$$\overline{U\tau}^{\langle US,\Gamma\rangle} \succ U\tau_1, \ldots, \overline{U\tau}^{\langle US,\Gamma\rangle} \succ U\tau_n,$$

are satisfied because $U(L[\tau/\alpha])$ holds. By the typing rule, this derivation is extended by

$$\langle US,\Gamma\rangle \vdash_{ML^+} \texttt{val rec}\, x = e : \{x : \overline{U\tau}^{\langle US,\Gamma\rangle}\}$$

to an ML^+-typing of $\texttt{val rec}\, x = e$ modulo Γ, which is represented by the final output,

$$\mathcal{W}^+_{dec}(\Gamma, \texttt{val rec}\, x = e) = (U(L[\tau/\alpha]), US, \{x : \overline{U\tau}^{\langle US,\Gamma\rangle}\}).$$

Remark. Phase 1 is like the first iteration of Mycroft's[20] iterative algorithm, except that he does not collect a set of constraints. He takes the derived type τ of e and uses $x : \overline{\tau}^{\langle S,\Gamma\rangle}$ as weakened assumption to retype e, etc.

3.3 The semi-algorithm $\mathcal{W}^+$ for inference of Milner-Mycroft types

Algorithm $\mathcal{W}^+$, which is stated in Figure 3, takes an expression e or declaration d and *two* environments Δ, Γ, with disjoint domain, as input. Δ contains the assumptions for monomorphic and Γ those for polymorphic free variables of e or d (i.e. those considered to be λ-bound resp. `val`- or `val rec`-bound in an enclosing term).

If $\mathcal{W}^+$ terminates, it returns *fail* or a triple (L, S, τ) resp. $(L, S, \{x : \overline{\tau}\})$, consisting of a semiunification problem L, a substitution S, and a type τ resp. a typing statement $\{x : \overline{\tau}\}$.

Inequality relations in L are indexed with 'fresh' indices that correspond to occurrences of polymorphic variables in e or d. Therefore, $\mathcal{W}^+$ has an implicit argument, a finite function $I : IVar \to 2^{I\!N}$, where $I(x) \subset I\!N$ enumerates the finitely many occurrences of x already visited; we assume $I(x) \cap I(y) = \emptyset$ for $x \neq y$. For each non-binding occurrence of x in e or d, $\mathcal{W}^+$ updates I by adding a new index to $I(x)$.

The constraints in L related to occurrences of polymorphic variables y in Γ form a subsystem used in the clause for function abstractions:

$$L^\Gamma := \bigcup \{\, L^y \mid y : \overline{\sigma} \in \Gamma \,\}, \quad \text{where} \quad L^y := \{\, \tau \sqsubseteq_i \rho \in L \mid i \in I(y) \,\}. \tag{6}$$

For the clauses of $\mathcal{W}^+_{dec}$, we need a definition. If a semiunification problem L in $\sqsubseteq_1, \ldots, \sqsubseteq_n$ holds, its set of *pattern variables* is

$$PV(L) := \bigcup_{1 \le i \le n} PV_i(L), \text{ where } PV_i(L) := \{\alpha \mid \tau \sqsubseteq_i \sigma \in L,\, \alpha \in FV(\tau),\, T_i(\alpha) \neq \alpha, \}.$$

The set of *specialized variables of L, relative to Γ*, written $spec(\Gamma, L)$, is the closure of $FV(\Gamma)$ under the relation R_L defined by

$$\alpha\ R_L\ \beta \iff \text{for some } 1 \le i \le n,\ \alpha \in PV_i(L) \text{ and } \beta \in T_i(\alpha).$$

By *mgsu* we mean a partial recursive function, which for unsolvable L may diverge or return *fail*, and for solvable L returns a most general semiunifier of L which does not trivially rename variables of L. Such *mgsu* exists by Theorem 6 and Proposition 7.

$\mathcal{W}^+$ is split into $\mathcal{W}^+_{exp}$ for expressions and $\mathcal{W}^+_{dec}$ for declarations:

- $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ x) =$
$$\begin{cases} (\emptyset, Id, \tau), & \text{if } \Delta(x) = \tau, \\ (L, Id, \tau[\alpha'/\alpha, \beta'/\beta]), & \text{if } \Gamma(x) = \forall\beta\tau, \text{ where} \end{cases}$$
$\alpha = FV(\forall\beta\tau) - FV(\Delta)$,
α' = new copies of α,
β' = new copies of β,
$L = \{\alpha \sqsubseteq_i \alpha' \mid \alpha \in \alpha\} \cup \{\delta \sqsubseteq_i \delta \mid \delta \in FV(\Delta)\}$,
with a fresh index i, and $I(x) := I(x) \cup \{i\}$

- $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ e_1 \cdot e_2) = (US_2L_1 \cup UL_2,\ US_2S_1,\ U\alpha)$
if $(L_1, S_1, \tau_1) = \mathcal{W}^+_{exp}(\Delta, \Gamma, e_1)$
$(L_2, S_2, \tau_2) = \mathcal{W}^+_{exp}(S_1\Delta, S_1\Gamma, e_2)$
$U = mgu(S_2\tau_1, \tau_2 \to \alpha)$, where α is a fresh variable.

- $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ (\texttt{fn}\, x \texttt{=>}\, e)) = (L, S,\ S\alpha \to \tau)$,
if $(L_\alpha, S, \tau) = \mathcal{W}^+_{exp}(\Delta, x : \alpha, \Gamma,\ e)$, where α is a fresh variable, and
$L := (L_\alpha - \{\, \tau' \sqsubseteq_i \tau' \mid \tau' \sqsubseteq_i \tau' \in L^\Gamma_\alpha \,\})$
$\cup\ \{\gamma \sqsubseteq_i \gamma \mid \tau' \sqsubseteq_i \tau' \in L^\Gamma_\alpha,\ \gamma \in FV(\tau') - (FV(S\alpha) - FV(S\Delta))\}$.

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

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

- $\mathcal{W}^+_{dec}(\Delta, \Gamma,\ \texttt{val rec}\, f \texttt{=}\, e) = (\tilde{U}(L[\tau/\alpha]), \tilde{U}S, \{f : \forall\beta\, \tilde{U}\tau\})$
if $(L, S, \tau) = \mathcal{W}^+_{exp}(\Delta, \Gamma, f : \alpha,\ e)$, with α fresh,
$\tilde{U} = mgsu(L[\tau/\alpha])$
$\beta = FV(\tilde{U}\tau) - FV(\tilde{U}S\Delta) - spec(\tilde{U}S\Gamma, \tilde{U}(L[\tau/\alpha]))$

- $\mathcal{W}^+_{exp}$ resp. $\mathcal{W}^+_{dec}$ returns *fail*,
if the call to *mgu* or *mgsu* or one of the recursive calls to $\mathcal{W}^+_{exp}$or $\mathcal{W}^+_{dec}$ returns *fail*.

Fig. 3. The type assignment function $\mathcal{W}^+$ for ML^+

Example 1. Consider a recursive declaration containing a non-recursive one.

```
val rec f = fn x => let val g = fn y => (x = f y; x) in g 0 end;
```

The local declaration of g is typed relative to $\Delta, \Gamma = \{x : \delta_x\}, \{f : \alpha_f\}$. In a first phase, we compute the type of the defining term $\lambda y.(x = fy; x)$. We assume $y : \delta_y$ and introduce an index $I(f) = \{1\}$ and a fresh variable α'_f for the occurrence of f to get:

$$\mathcal{W}^+_{exp}(\{x : \delta_x, y : \delta_y\}, \{f : \alpha_f, g : \alpha_g\}, f) = \\ (\{\alpha_f \sqsubseteq_1 \alpha'_f, \delta_x \sqsubseteq_1 \delta_x, \delta_y \sqsubseteq_1 \delta_y\}, Id, \alpha'_f).$$

The application fy then types as

$$\mathcal{W}^+_{exp}(\{x : \delta_x, y : \delta_y\}, \{f : \alpha_f, g : \alpha_g\}, fy) = \\ (\{\alpha_f \sqsubseteq_1 \delta_y \to \beta, \delta_x \sqsubseteq_1 \delta_x, \delta_y \sqsubseteq_1 \delta_y\}, [\delta_x \to \beta/\alpha'_f], \beta).$$

Assuming that $=$ and ; are infix operators of type $\forall\alpha\forall\beta\,(\alpha \times \alpha \to \texttt{bool})$ and $\forall\alpha\forall\beta\,(\alpha \times \beta \to \beta)$, respectively, we get

$$\mathcal{W}^+_{exp}(\{x : \delta_x, y : \delta_y\}, \{f : \alpha_f, g : \alpha_g\}, (x = fy; x)) = \\ (\{\alpha_f \sqsubseteq_1 \delta_y \to \delta_x, \delta_x \sqsubseteq_1 \delta_x, \delta_y \sqsubseteq_1 \delta_y\}, [\delta_x/\beta][\delta_x \to \beta/\alpha'_f], \delta_x).$$

When typing $\lambda y.(x = fy; x)$, the constraint $\delta_y \sqsubseteq_1 \delta_y$ is withdrawn, since y now is known to have smaller scope than the open polymorphic variables f, g:

$$\mathcal{W}^+_{exp}(\{x : \delta_x\}, \{f : \alpha_f, g : \alpha_g\}, \lambda y.(x = fy; x)) = \\ (\{\alpha_f \sqsubseteq_1 \delta_y \to \delta_x, \delta_x \sqsubseteq_1 \delta_x\}, [\delta_x/\beta][\delta_x \to \beta/\alpha'_f], \delta_y \to \delta_x).$$

The second phase of typing the declaration of g is to generalize the type $\delta_y \to \delta_x$ of its defining expression $\lambda y.(x = fy; x)$ properly. Since δ_x and δ_y occur in the constraint set in a specialization of the polytype assumption $f : \alpha_f$, they must not be quantified. Hence

$$\mathcal{W}^+_{dec}(\{x : \delta_x\}, \{f : \alpha_f\}, \texttt{val}\ g = \lambda y.(x = fy; x)) = \\ (\{\alpha_f \sqsubseteq_1 \delta_y \to \delta_x, \delta_x \sqsubseteq_1 \delta_x\}, [\delta_x/\beta][\delta_x \to \beta/\alpha'_f], \{g : \delta_y \to \delta_x\}). \tag{7}$$

The extended environment

$$\Delta, \Gamma' = \{x : \delta_x\}, \{f : \alpha_f, g : \delta_y \to \delta_x\}$$

is now used to type the remaining body $g\,0$ of the definition of f. Since g is to be polymorphic, we put $I(g) := \{2\}$ and add new constraints as in

$$\mathcal{W}^+_{exp}(\Delta, \Gamma', g) = (\{\delta_y \sqsubseteq_2 \delta'_y, \delta_x \sqsubseteq_2 \delta_x\}, Id, \delta'_y \to \delta_x),$$

obtaining

$$\mathcal{W}^+_{exp}(\Delta, \Gamma', g0) = (\{\delta_y \sqsubseteq_2 int, \delta_x \sqsubseteq_2 \delta_x\}, [int/\delta'_y], \delta_x). \tag{8}$$

By collecting the constraints and composing the substitutions of (7) and (8) to

$$L = \{\alpha_f \sqsubseteq_1 \delta_y \to \delta_x, \delta_x \sqsubseteq_1 \delta_x\} \cup \{\delta_y \sqsubseteq_2 int, \delta_x \sqsubseteq_2 \delta_x\},$$
$$S = [int/\delta'_y][\delta_x/\beta][\delta_x \to \beta/\alpha'_f],$$

we therefore get

$$\mathcal{W}^+_{exp}(\Delta, \Gamma, \texttt{let val } g = \lambda y.(x = fy; x) \texttt{ in } g0 \texttt{ end}) = (L, S, \delta_x).$$

When discharging the assumption $x : \delta_x$ to type the defining term for f, the constraint $\delta_x \sqsubseteq_i \delta_x$ for occurrences i of each assumption in Γ, i.e. for $i \in I(f)$, has to be removed:

$$\mathcal{W}^+_{exp}(\emptyset, \Gamma, \lambda x.\texttt{let val } g = \lambda y.(x = fy; x) \texttt{ in } g0 \texttt{ end})$$
$$= (\{\alpha_f \sqsubseteq_1 \delta_y \to \delta_x, \delta_y \sqsubseteq_2 int, \delta_x \sqsubseteq_2 \delta_x\}, S, \delta_x \to \delta_x)$$

In the second phase of typing f, we have to replace α_f by the derived type $\delta_x \to \delta_x$ in the constraints and solve the resulting system

$$\{\delta_x \to \delta_x \sqsubseteq_1 \delta_y \to \delta_x,\ \delta_y \sqsubseteq_2 int,\ \delta_x \sqsubseteq_2 \delta_x\}. \tag{9}$$

A most general semiunifier U must satisfy $U(\delta_y) = T_1U(\delta_x) = U(\delta_x)$ by the first inequation, and then $U(\delta_x) = T_2U(\delta_x) = T_2U(\delta_y) = int$ by the second and third. Hence $U = [int/\delta_x, int/\delta_y]$ and

$$\mathcal{W}^+_{dec}(\emptyset, \emptyset, \texttt{val rec } f = \texttt{let} \ldots \texttt{in} \ldots \texttt{end}) = (UL, US, \{f : int \to int\}).$$

Note that in typing λ-abstractions, the constraints $\delta_x \sqsubseteq_i \delta_x$ must not be removed for occurrences i of variables already discharged: would $\delta_x \sqsubseteq_2 \delta_x$ be missing in (9), we would infer $f : \forall\delta_x(\delta_x \to \delta_x)$. But this is not an ML^+-type of the declared variable, since

$$f : \forall\delta_x(\delta_x \to \delta_x) \not\vdash_{ML+} (\texttt{fn } x \texttt{ => let} \ldots \texttt{in} \ldots \texttt{end}) : \delta_x \to \delta_x.$$

Note also that it would have been wrong to quantify δ_y in the type of g in (7): in the derivation finally obtained, $\delta_y = int$.

4 Correctness and weak completeness of $\mathcal{W}^+$

We will now show that $\mathcal{W}^+$ is sound, weakly complete, and computes principal ML^+-typings. To define principality, we first introduce a notion of ML^+-typing *modulo* an environment that allows certain polytype assumptions to be weakened. This is needed to inductively prove a principal types property for recursive declarations, where the typing obtained in phase 1 has to be weakend in phase 2.

Think of the input Δ, Γ of $\mathcal{W}^+$ as the environment $\Delta, \overline{\Gamma}^\Delta = \Delta \cup \overline{\Gamma}^\Delta$ plus an information that type quantifiers in $\overline{\Gamma}^\Delta$ binding variables $\alpha \in FV(\Gamma) - FV(\Delta)$ may be weakend:

Definition 9. We call (S,τ) a *typing of expression e modulo Δ,Γ* if $S\Delta, \overline{S\Gamma}^{S\Delta} \vdash_{ML^+} e:\tau$, and $(S,\{y:\overline{\tau}\})$ a *typing of declaration d modulo Δ,Γ* if $S\Delta, \overline{S\Gamma}^{S\Delta} \vdash_{ML^+} d:\{y:\overline{\tau}\}$. The typing (S,τ) of e modulo Δ,Γ is *principal*, if for any typing (S',τ') of e modulo Δ,Γ there is a substitution U such that $\tau' = U\tau$ and $S' = US$ on the free variables of Δ,Γ.

Typings (S,τ) of e modulo Δ,Γ represent ML^+-typability statements with respect to the environment $S\Delta, \overline{S\Gamma}^{S\Delta}$, which is *not* just the instantiation by S of free (mono)type variables of $\Delta, \overline{\Gamma}^{\Delta}$.[7] Additionally, for $y:\overline{\sigma}$ in Γ, $\overline{S\Gamma}^{S\Delta}$ replaces the quantified assumption $y:\overline{\overline{\sigma}}^{\Delta}$ of $\overline{\Gamma}^{\Delta}$ by $y:\overline{S\overline{\sigma}}^{S\Delta}$ via instanting bound quantifiers of $\overline{\overline{\sigma}}^{\Delta}$ not already in $\overline{\sigma}$.

Theorem 10. *Suppose Δ,Γ contains a type assumption for each free individual variable of the expression e resp. declaration d. Assume that for different assumptions $y_1:\overline{\sigma}_1, y_2:\overline{\sigma}_2$ in Γ, $FV(\overline{\sigma}_1)\cap FV(\overline{\sigma}_2)\subseteq FV(\Delta)$. Then*

(a) If $\mathcal{W}^+_{exp}(\Delta,\Gamma,e) = (L,S,\tau)$, then

1. *Id is a semiunifier of L,*
2. *for all (S',τ') the following are equivalent:*
   - *(i) (S',τ') is a typing of e modulo Δ,Γ,*
   - *(ii) for some semiunifier U of L, $S' =_{\Delta,\Gamma} US$, and $\tau' = U\tau$.*
3. *For different $y_1:\overline{\sigma}_1, y_2:\overline{\sigma}_2$ in Γ, $FV(S\overline{\sigma}_1)\cap FV(S\overline{\sigma}_2)\subseteq FV(S\Delta)$.*

(b) If $\mathcal{W}^+_{dec}(\Delta,\Gamma,d) = (L,S,\{x:\overline{\tau}\})$ and $\overline{\tau} = \forall\beta\tau$, then

1. *Id is a semiunifier of L,*
2. *for each $(S',\{x:\overline{\sigma}\})$, the following are equivalent:*
   - *(i) $(S',\{x:\overline{\sigma}\})$ is a typing of d modulo Δ,Γ,*
   - *(ii) for some semiunifier U of L, $S' =_{\Delta,\Gamma} US$ and $\overline{\sigma} = \overline{U\tau}^{US\Delta}$,*
3. *For different $y_1:\overline{\sigma}_1, y_2:\overline{\sigma}_2$ in Γ, $FV(S\overline{\sigma}_1)\cap FV(S\overline{\sigma}_2)\subseteq FV(S\Delta)$; moreover, $FV(\overline{\tau})\cap FV(S\overline{\sigma}_1)\subseteq FV(S\Delta)$.*

(c) If $\mathcal{W}^+_{exp}(\Delta,\Gamma,e) =$ fail or does not terminate, then e is not ML^+-typable modulo Δ,Γ. Similarly for $\mathcal{W}^+_{dec}(\Delta,\Gamma,d)$.

By Claim 1 and Claim 2, (ii) $\Rightarrow$ (i), with $U = Id$, one obtains $S\Delta, \overline{S\Gamma}^{S\Delta} \vdash_{ML^+} e:\tau$ from $\mathcal{W}^+_{exp}(\Delta,\Gamma,e) = (L,S,\tau)$. Claim 2, (i) $\Rightarrow$ (ii), says that the typing produced by $\mathcal{W}^+_{exp}$ is the principal ML^+-typing of e modulo Δ,Γ. Any other typing of e modulo Δ,Γ is obtained by applying a semiunifier of the deliverd constraint set L. Similarly for $\mathcal{W}^+_{dec}$ and d. Claim 3 says

[7] Since S may be defined on $FV(\Gamma) - FV(\Delta)$, $\overline{S\Gamma}^{S\Delta}$ cannot be written as $S(\overline{\Gamma}^{\Delta})$, for which, by convention, we assumed that bound variables of $\overline{\Gamma}^{\Delta}$ do not occur in the domain and range of S.

the additional hypothesis that types of polymorphic assumptions are related via monotype parameters only is preserved, so that $\mathcal{W}^+$ can be used recursively.

For ML^+-untypable recursive declarations, $\mathcal{W}^+_{dec}(\Delta, \Gamma, \texttt{val rec}\, x = e)$ need not terminate, since the computation of most general semiunifiers may diverge.

The proof of Theorem 10, like the one for Theorem 8, envolves showing that the combination of substitutions delivered by recursive calls of $\mathcal{W}^+$ corresponds to combinations and 'specializations' of ML^+-derivations. Since 'specializing' substitutions now may also weaken polytype assumptions, the proof of (ii) $\Rightarrow$ (i) implicitly shows a strengthening of Lemma 1:

Lemma 11. *If $\Delta, \overline{\Gamma}^{\Delta} \vdash_{ML^+} e : \tau$, there is a semiunification problem L solved by Id, such that $S\Delta, \overline{S\Gamma}^{S\Delta} \vdash_{ML^+} e : S\tau$ for any semiunifier S of L.*

For S not satisfying the constraints, the induced weakening of assumptions may lead to underivable typing statements:

Example 2. In example 1, an intermediate stage represented the ML^+-typing

$$\{x : \delta_x, f : \forall \alpha_f.\alpha_f, g : \forall \alpha_g.\alpha_g\} \vdash_{ML^+} \lambda y.(x = fy; x) : \delta_y \to \delta_x.$$

Clearly, if $U(\alpha_f) = int$, the constraints $L = \{\alpha_f \sqsubseteq_1 \delta_y \to \delta_x, \delta_x \sqsubseteq_1 \delta_x\}$ given by $\mathcal{W}^+_{exp}$ are not satisfied, and indeed

$$\{x : \delta_x, f : int, g : \forall \alpha_g.\alpha_g\} \not\vdash_{ML^+} \lambda y.(x = fy; x) : \delta_y \to \delta_x.$$

Observe that $\mathcal{W}^+$ delivers a solved constraint set for each expression or declaration, but solves semiunification constraints only in the case of recursive declarations. To combine two subderivations, we therefore need that the constraint set L of the first remains solved when the specializing substitution S of the second is applied to it. By remark 1 on p.8, SL might be unsolvable for arbitrary S; but substitutions arising during type checking are of a certain kind and *do* preserve the solvedness of previously given constraints (cf. Lemma 12 and section 5.1).

Proof of Theorem 10, by induction on the expression or declaration being typed.

We will show a few further properties, relating the input environments Δ, Γ and the triple (L, S, τ) resp. $(L, S, \{x : \overline{\tau}\})$ returned by $\mathcal{W}^+$. To state these, we need two more definitions.

A substitution S *affects* the variable α if $S\alpha \neq \alpha$ or $\alpha \in FV(S\beta)$ for some β with $S\beta \neq \beta$.

We say "$L \Vdash S\Delta \sqsubseteq_i S\Delta$", iff for each $\delta \in FV(S\Delta)$, there is some type τ' with $\tau' \sqsubseteq_i \tau' \in L$ and $\delta \in FV(\tau')$.

The additional claims, proven simultaneously with claims 1.-3., are the following:

0. i) S does not affect variables of $FV(\Gamma) - FV(\Delta)$,
ii) $FV(rhs(L), \tau) \cap FV(S\Gamma) \subseteq FV(S\Delta)$,
iii) (a) $PV(L) \cap FV(S\Delta, \tau) = \emptyset$ resp. (b) $PV(L) \cap FV(S\Delta) = \emptyset$,

iv) For each $\sqsubseteq_i$ *in* L, $L \Vdash S\Delta \sqsubseteq_i S\Delta$,

v) For each $y : \overline{\sigma} \in \Gamma$, $PV(L^y) \subseteq FV(S\overline{\sigma})$,

vi) For each $y : \overline{\sigma} \in \Gamma$, $FV(L - L^y) \cap FV(S\overline{\sigma}) \subseteq FV(S\Delta)$.

Remark 2. The meaning of these conditions roughly is as follows:

(i) The polytype unknowns $\alpha \in FV(\Gamma) - FV(\Delta)$ of the input represent quantified type variables, and the corresponding quantified assumption (of free polymorphic individual variables) is not weakened while inferring a type of the input expression. (Weakening of quantified assumptions only occurs before discharging a polytype assumption for a recursion variable, and for already discharged assumptions.)

(ii) Type variables in the derived type τ (resp. body of the derived typing $x : \overline{\tau}$) or in types of occurrences of polymorphic variables (recorded in $rhs(L)$) must be monotypes, if they occur in the refined environment.

(iii) The derived type τ of an expression does not contain type variables known to represent polytypes; this cannot be demanded for derived typings for a declaration.

(iv) L syntactically enforces "$PV(UL) \cap FV(US\Delta) = \emptyset$ for all semiunifiers U of L"; with claim 1, most of (iii) follows:

$$PV(L) \cap FV(S\Delta) = \emptyset. \tag{10}$$

Stated less formally, allowed specializations of the (implicitly constructed) typing derivation of the current input will not instantiate monotype parameters in $S\Delta$ so that they contain polytype unknowns.

(v+vi) Polytype unknowns of $FV(S\Gamma) - FV(S\Delta)$ occur in L *only* on the left hand sides of inequation relations of the subsystem $L^\Gamma \subseteq L$. More precisely, after typing an expression or declaration, for each $y : \overline{\sigma} \in \Gamma$, we have

$$PV(L^y) \cap FV(L - L^y) = \emptyset = PV(L^y) \cap FV(rhs(L)). \tag{11}$$

The first equation follows from a) 0.(v), 0.(vi) and 0.(iii), the second from (a) 0.(v), 0.(ii) and 0.(iii).

Proof of claim 3: For expressions, it follows from 0.(i). For declarations, note that 3. holds for assumptions in Γ by (b) 0.(i), and by (b) 0.(ii) it also holds for the new assumption $x : \overline{\tau}$.

Next we prove some lemmata on semiunification that are common to various cases.

Lemma 12. *Let* L *be a semiunification problem and* S *a substituion. If* L *holds and* S *does not affect pattern variables of* L*, then* SL *also holds and* $PV(SL) = PV(L)$.

Proof Let T_i be matching substitutions so that $(Id, T_1, \ldots, T_n)$ is a solution of L. Define

$$T_i'\gamma := \begin{cases} ST_i\gamma, & \text{if } \gamma \in PV(L), \\ \gamma, & \text{else.} \end{cases}$$

Pick $\sigma \sqsubseteq_i \tau \in L$ where $\sigma = \sigma(\boldsymbol{\alpha}, \boldsymbol{\beta})$ with $\boldsymbol{\beta} = FV(\sigma) \cap PV(L)$.

$$\begin{aligned} T_i' S\sigma &= \sigma(T_i' S\boldsymbol{\alpha}, T_i' S\boldsymbol{\beta}) \\ &= \sigma(S\boldsymbol{\alpha}, T_i'\boldsymbol{\beta}) && \text{since } FV(S\boldsymbol{\alpha}) \cap PV(L) = \emptyset, \\ &= \sigma(S\boldsymbol{\alpha}, ST_i\boldsymbol{\beta}) \\ &= ST_i\sigma && \text{since } \boldsymbol{\alpha} \cap PV(L) = \emptyset \\ &= S\tau. \end{aligned}$$

Hence $(Id, T_1', \ldots, T_n')$ is a solution of SL. By definition, if $T_i'\gamma \neq \gamma$, then $\gamma \in PV(L)$. Conversely, if $\gamma \in PV(L)$, then $T_i\gamma \neq \gamma$ for some i, and since S does not affect pattern variables of L we must have $T_i'\gamma = ST_i\gamma \neq \gamma$. □

The condition that S must not have pattern variables of L in its *range* excludes counterexamples like the following: $L = \{\alpha \to \beta \sqsubseteq \alpha \to \alpha\}$ has β as pattern variable and is solved by $(Id, [\alpha/\beta])$, but for $S = [\beta \to \beta/\alpha]$, $SL = \{(\beta \to \beta) \to \beta \sqsubseteq (\beta \to \beta) \to (\beta \to \beta)\}$ is unsolvable.

Proposition 13. *Suppose S_1 does not affect variables of $FV(\Gamma) - FV(\Delta)$.*

(i) If S_2 does not affect $FV(S_1\Gamma) - FV(S_1\Delta)$, then S_2S_1 does not affect variables of $FV(\Gamma) - FV(\Delta)$.

(ii) If $FV(\tau) \cap FV(\Gamma) \subseteq FV(\Delta)$, then $FV(S_1\tau) \cap FV(S_1\Gamma) \subseteq FV(S_1\Delta)$.

Proof (i) Suppose $\alpha \in FV(\Gamma) - FV(\Delta)$. Note that then $\alpha \notin FV(S_1\Delta)$, because otherwise there were $\beta \in FV(\Delta)$ with $\alpha \in FV(S_1\beta)$. But since $\alpha \neq \beta$ this showed $\beta \neq S_1\beta$, and thus S_1 would affect α.

Claim 1: $S_2S_1\alpha = \alpha$. Since S_1 does not affect α, we have $S_1\alpha = \alpha$ and hence $\alpha \in FV(S_1\Gamma)$. Because $\alpha \notin FV(S_1\Delta)$, S_2 does not affect α and so $S_2S_1\alpha = S_2\alpha = \alpha$.

Claim 2: For no β with $S_2S_1\beta \neq \beta$ is $\alpha \in FV(S_2S_1\beta)$. Suppose we have such a β. If $\alpha \notin FV(S_1\beta)$, there is $\alpha \neq \gamma \in FV(S_1\beta)$ with $\alpha \in FV(S_1\gamma)$. Since S_1 does not affect α, we must have $S_1\gamma = \gamma = \alpha$, a contradiction. So $\alpha \in FV(S_1\beta)$, and hence $\beta = S_1\beta = \alpha$, because S_1 does not affect α. So $\alpha = S_1\alpha$ and $S_2S_1\alpha \neq \alpha$, contradicting claim 1. It follows that the assumed β cannot exist.

(ii) Suppose $\beta \in FV(S_1\alpha) \cap FV(S_1\gamma)$ for $\alpha \in FV(\tau)$ and $\gamma \in FV(\Gamma)$. Since S_1 does not affect $FV(\Gamma) - FV(\Delta)$, either $\gamma \in FV(\Delta)$, hence $\beta \in FV(S_1\Delta)$, or else $S_1\gamma = \gamma = \beta$ and then $\beta \in FV(\Gamma) \cap FV(S_1\alpha)$ gives $\beta = \alpha \in FV(\tau) \cap FV(\Gamma) \subseteq FV(\Delta)$, because $\beta = \gamma$ is not affected by S_1. But $\beta \in FV(\Delta)$ also gives $\beta = S_1\beta \in FV(S_1\Delta)$. □

The following two lemmata are needed for the case of recursive declarations.

Lemma 14. *Let U be a most general semiunifier of L that does not trivially rename variables of L. Then U does not affect variables of L occurring only on left hand sides of inequations.*

Proof Suppose $\gamma \in FV(L)$ occurs only on left hand sides of inequations. Expand U to a most general solution $(U, T_1, \ldots, T_n)$ of L. If $U\gamma$ is not a variable, there is a strictly more

general solution, defined with a fresh variable $\tilde{\gamma}$:

$$\tilde{U}\alpha := \begin{cases} \tilde{\gamma}, & \text{if } \alpha = \gamma \\ U\alpha, & \text{else} \end{cases}, \qquad \tilde{T}_i\alpha := \begin{cases} T_iU\gamma, & \text{if } \alpha = \tilde{\gamma} \\ T_i\alpha, & \text{else.} \end{cases}$$

So let $U\gamma$ be the variable γ', and suppose γ is affected by U.

Case 1: $\gamma' = \gamma \in FV(U\delta)$ for some $\delta \neq U\delta$. Then $\delta \neq \gamma = U\gamma$. Let $\tilde{\gamma}$ be fresh and define $(\tilde{U}, \tilde{T}_1, \ldots, \tilde{T}_n)$ by

$$\tilde{U}\alpha := \begin{cases} \alpha, & \text{if } \alpha = \gamma \\ (U\alpha)[\tilde{\gamma}/\gamma], & \text{else} \end{cases}, \qquad \tilde{T}_i\alpha := \begin{cases} (T_i\gamma)[\tilde{\gamma}/\gamma], & \text{if } \alpha = \tilde{\gamma}, \\ (T_i\alpha)[\tilde{\gamma}/\gamma], & \text{if } \alpha \neq \tilde{\gamma} \end{cases}$$

This is another solution of L: If $\rho \sqsubseteq_i \sigma \in L$, where $\rho = \rho(\alpha, \gamma)$ and $U\alpha = \tau(\beta, \gamma)$ for simplicity, then

$$\begin{aligned}
\tilde{U}\sigma &= (U\sigma)[\tilde{\gamma}/\gamma] && \text{since } \gamma \notin FV(\sigma), \\
&= (T_iU\rho)[\tilde{\gamma}/\gamma] && \text{since } (U, T_1, \ldots, T_n) \text{ solves } \rho \sqsubseteq_i \sigma, \\
&= \rho(\tau(T_i\beta, T_i\gamma)[\tilde{\gamma}/\gamma], (T_iU\gamma)[\tilde{\gamma}/\gamma]) \\
&= \rho(\tau(\tilde{T}_i\beta, \tilde{T}_i\tilde{\gamma}), \tilde{T}_i\gamma) && \text{since } (T_iU\gamma)[\tilde{\gamma}/\gamma] = (T_i\gamma)[\tilde{\gamma}/\gamma] = \tilde{T}_i\tilde{\gamma} = \tilde{T}_i\gamma, \\
&= \rho(\tilde{T}_i((U\alpha)[\tilde{\gamma}/\gamma]), \tilde{T}_i\tilde{U}\gamma) \\
&= \rho(\tilde{T}_i\tilde{U}\alpha, \tilde{T}_i\tilde{U}\gamma) \\
&= \tilde{T}_i\tilde{U}\rho.
\end{aligned}$$

But, contradicting the assumption, U is not as general as $\tilde{U}$: if $\tilde{U} = SU$ on $FV(L)$, we have $S\gamma = SU\gamma = \tilde{U}\gamma = \gamma$, so that $\gamma \in FV(SU\delta) = FV(\tilde{U}\delta)$ since $\gamma \in FV(U\delta)$. But on the other hand, $\gamma \notin FV((U\delta)[\tilde{\gamma}/\gamma]) = FV(\tilde{U}\delta)$.

Case 2: $U\gamma = \gamma' \neq \gamma$. Since U does not trivially rename γ, we have $\gamma' \in FV(U\delta)$ for some $\delta \neq \gamma$ in L. With fresh $\tilde{\gamma}$, define $(\tilde{U}, \tilde{T}_1, \ldots, \tilde{T}_n)$ as above, except that $\tilde{T}_i\gamma := (T_i\gamma')[\tilde{\gamma}/\gamma]$. From

$$(T_i\gamma)[\tilde{\gamma}/\gamma] = \tilde{T}_i\tilde{\gamma} \quad \text{and} \quad (T_iU\gamma)[\tilde{\gamma}/\gamma] = (T_i\gamma')[\tilde{\gamma}/\gamma] = \tilde{T}_i\gamma,$$

we again obtain $\tilde{U}\sigma = (T_iU\rho)[\tilde{\gamma}/\gamma] = \tilde{T}_i\tilde{U}\rho$. Suppose $\tilde{U} = SU$ on $FV(L)$. Then $S\gamma' = SU\gamma = \tilde{U}\gamma = \gamma$, whence all occurrences of γ' in $U\delta$ turn into occurrences of γ in $SU\delta = \tilde{U}\delta$. But since $\gamma' \in FV(U\delta)$ and $\gamma \neq \gamma'$, there is an occurrence of γ' in $U\delta$ that is unchanged when going to $(U\delta)[\tilde{\gamma}/\gamma] = \tilde{U}\delta$. Again, U would not be a most general semiunifier of L. □

Lemma 15. *Let L be a semiunification problem with $L \Vdash S\Delta \sqsubseteq_i S\Delta$ for each $\sqsubseteq_i$ in L, and suppose Id is a semiunifier of L and τ a type. For each semiunifier U of L there is a semiunifier U' of L such that, up to renaming of bound quantifiers,*

$$\overline{U'\tau}^{U'S\Delta} = \overline{U\forall\boldsymbol{\beta}\tau}^{US\Delta}, \text{ where } \boldsymbol{\beta} = FV(\tau) - FV(S\Delta) - spec(S\Gamma, L).$$

Proof Define U' by

$$U'\alpha = \begin{cases} U\alpha, & \text{if } \alpha \in FV(S\Delta) \cup spec(S\Gamma, L), \\ \alpha' & \text{for some fresh variable } \alpha' \notin FV(US\Delta, Uspec(S\Gamma, L)), \text{ else.} \end{cases}$$

For suitable residual substitutions, $(Id, \tilde{T}_1, \ldots, \tilde{T}_n)$ and $(U, T_1, \ldots, T_n)$ are solutions of L. To show that U' is a semiunifier of L, we have to find matching substitutions $T'_1, \ldots, T'_n$ such that

$$T'_i U'\sigma = U'\rho \quad \text{for each } \sigma \sqsubseteq_i \rho \text{ in } L.$$

Let T'_i be the restriction of T_i to $FV(US\Delta, U\mathit{spec}(S\Gamma, L))$, extended to the fresh variables by $T'_i\beta' = U'\tilde{T}_i\beta$ for $\beta \in \boldsymbol{\beta}$. Pick $\sigma \sqsubseteq_i \rho \in L$. By choice of T_i and $\tilde{T}_i$,

$$\tilde{T}_i =_{S\Delta} Id,\ \rho = \tilde{T}_i\sigma, \quad \text{and} \quad T_i =_{US\Delta} Id,\ U\rho = T_iU\sigma. \tag{12}$$

Let $\sigma = \sigma(\boldsymbol{\alpha}, \boldsymbol{\beta})$, where $\boldsymbol{\beta} = FV(\sigma) - FV(S\Delta) - \mathit{spec}(S\Gamma, L)$ and $\boldsymbol{\alpha} = FV(\sigma) - \boldsymbol{\beta}$. By (12), $U'\rho = U'\tilde{T}_i\sigma$, so the claim $T'_iU'\sigma = U'\rho$ amounts to

$$T'_iU'\boldsymbol{\alpha} = U'\tilde{T}_i\boldsymbol{\alpha} \quad \text{and} \quad T'_iU'\boldsymbol{\beta} = U'\tilde{T}_i\boldsymbol{\beta}.$$

Let $\alpha \in \boldsymbol{\alpha}$. If $\alpha \in \mathit{spec}(S\Gamma, L)$, then $FV(\tilde{T}_i\alpha) \subseteq \mathit{spec}(S\Gamma, L,)$ and hence $U'\tilde{T}_i\alpha = U\tilde{T}_i\alpha$. If $\alpha \in FV(S\Delta)$, then $\tilde{T}_i\alpha = \alpha$ and hence also $U'\tilde{T}_i\alpha = U\tilde{T}_i\alpha$. Since T'_iU' agrees with T_iU on the free variables of $S\Delta, \mathit{spec}(S\Gamma, L)$ and, by (12), $T_iU\alpha = U\tilde{T}_i\alpha$ from $T_iU\sigma = U\rho = U\tilde{T}_i\sigma$, this gives

$$T'_iU'\boldsymbol{\alpha} = T_iU\boldsymbol{\alpha} = U\tilde{T}_i\boldsymbol{\alpha} = U'\tilde{T}_i\boldsymbol{\alpha}.$$

On the other hand, for $\beta \in \boldsymbol{\beta}$ by choice of $\tilde{T}_i$ we also have

$$T'_iU'\beta = T'_i\beta' = \tilde{U}'\tilde{T}_i\beta.$$

Hence U' is a semiunifier of L.

For the second claim, let $\boldsymbol{\alpha} = FV(\tau) - \boldsymbol{\beta}$. Since $\boldsymbol{\beta}' \notin FV(US\Delta) = FV(U'S\Delta)$,

$$\begin{aligned}\overline{U'\tau}^{U'S\Delta} &= \overline{\tau(U'\boldsymbol{\alpha}, U'\boldsymbol{\beta})}^{U'S\Delta} = \overline{\tau(U\boldsymbol{\alpha}, \boldsymbol{\beta}')}^{U'S\Delta} \\ &= \overline{\forall\boldsymbol{\beta}'.\, \tau(U\boldsymbol{\alpha}, \boldsymbol{\beta}')}^{US\Delta} = \overline{U(\forall\boldsymbol{\beta}\,\tau)}^{US\Delta},\end{aligned}$$

by definition of $U(\forall\boldsymbol{\beta}\,\tau)$. □

We are now ready to show that $\mathcal{W}^+$ computes principal types for ML^+-typable expressions and declarations.

Case $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ x) =$

$$\begin{cases} (\emptyset, Id, \tau), & \text{if } \Delta(x) = \tau, \\ (L, Id, \tau[\boldsymbol{\alpha}'/\boldsymbol{\alpha}, \boldsymbol{\beta}'/\boldsymbol{\beta}]), & \text{if } \Gamma(x) = \forall\boldsymbol{\beta}\tau, \text{ where} \\ \quad \boldsymbol{\alpha} = FV(\forall\boldsymbol{\beta}\tau) - FV(\Delta), \\ \quad \boldsymbol{\alpha}' = \text{new copies of } \boldsymbol{\alpha}, \\ \quad \boldsymbol{\beta}' = \text{new copies of } \boldsymbol{\beta}, \\ \quad L = \{\alpha \sqsubseteq_i \alpha' \mid \alpha \in \boldsymbol{\alpha}\} \cup \{\delta \sqsubseteq_i \delta \mid \delta \in FV(\Delta)\}, \\ \quad \text{with a new relation } \sqsubseteq_i, \text{ and } i \text{ added to } I(x) \end{cases}$$

0. If $x : \tau \in \Delta$, the claims are empty or trivial. If $x : \forall \boldsymbol{\beta}\, \tau \in \Gamma$, we have:

   (i) Id does not affect variables at all.

   (ii) By the choice of $\boldsymbol{\alpha}', \boldsymbol{\beta}'$, $rhs(L)$ and $\tau[\boldsymbol{\alpha}'/\boldsymbol{\alpha}, \boldsymbol{\beta}'/\boldsymbol{\beta}]$ contain fresh variables and variables of $FV(\Delta)$ only.

   (iii) Variables in $PV(L) = \{\boldsymbol{\alpha}\}$ do not occur in $\tau[\boldsymbol{\alpha}'/\boldsymbol{\alpha}, \boldsymbol{\beta}'/\boldsymbol{\beta}]$ or $FV(\Delta)$.

   (iv) Since $\sqsubseteq_i$ is the only inequation relation in L, $\{\, \delta \sqsubseteq_i \delta \mid \delta \in FV(\Delta) \,\} \subseteq L$ ensures the claim.

   (v) For $y : \overline{\sigma} \in \Gamma$ with $y \neq x$, we have $L^y = \emptyset \subseteq FV(\overline{\sigma})$, and when $y = x$, we have $PV(L^x) = \{\boldsymbol{\alpha}\} \subseteq FV(\forall \boldsymbol{\beta}\, \tau)$.

   (vi) For $y : \overline{\sigma} \in \Gamma$ with $y \neq x$, we have

$$FV(L - L^y) \cap FV(\overline{\sigma}) \subseteq \{\boldsymbol{\alpha}, \boldsymbol{\alpha}'\} \cap FV(\overline{\sigma}) \subseteq FV(\Delta),$$

   since $FV(\forall \boldsymbol{\beta}\, \tau) \cap FV(\overline{\sigma}) \subseteq FV(S\Delta)$, by the assumption about Δ, Γ. For $y = x$, the claim is trivial since $L - L^x = \emptyset$.

1. (Id, T_i) is a solution of L, where $T_i(\alpha) = \alpha'$ for each $\alpha \in \boldsymbol{\alpha}$, and $T_i(\delta) = \delta$ for $\delta \in FV(\Delta)$.

2. The case where $x : \tau$ is in Δ is left to the reader. Consider $x : \forall \boldsymbol{\beta}.\tau \in \Gamma$, where $\tau = \tau(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\delta})$ with $\boldsymbol{\delta} = FV(\Delta)$.

   (ii) $\Rightarrow$ (i): Let U be a semiuinfier of L, S' a substitution with $S' =_{\Delta,\Gamma} UId$ and $\tau' = U(\tau[\boldsymbol{\alpha}'/\boldsymbol{\alpha}, \boldsymbol{\beta}'/\boldsymbol{\beta}])$. Then[8]

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta}(x) = \overline{U\Gamma(x)}^{U\Delta} = \overline{U\forall\boldsymbol{\beta}\tau}^{U\Delta} = \overline{\forall\boldsymbol{\beta} U\tau}^{U\Delta}.$$

   Choose T_i such that (U, T_i) solves L, and let T_i' be T_i extended by $[U\boldsymbol{\beta}'/\boldsymbol{\beta}]$. Since $T_iU\boldsymbol{\alpha} = U\boldsymbol{\alpha}'$, $T_iU\boldsymbol{\delta} = U\boldsymbol{\delta}$ and $T_i'U\boldsymbol{\beta} = T_i'\boldsymbol{\beta} = U\boldsymbol{\beta}'$, we get

$$T_i'U\tau = T_i'U(\tau(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\delta})) = U(\tau(\boldsymbol{\alpha}', \boldsymbol{\beta}', \boldsymbol{\delta})) = \tau',$$

   so $\overline{\forall\boldsymbol{\beta}\, U\tau}^{U\Delta} \succ \tau'$, since $dom(T_i') \subseteq \boldsymbol{\beta} \cup (FV(U\tau) - FV(U\Delta))$. It follows that $S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash x : \tau'$.

   (i) $\Rightarrow$ (ii): Let (S', τ') be a typing modulo Δ, Γ. Then $S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash x : \tau'$ and

$$\overline{S'\Gamma(x)}^{S'\Delta} = \overline{S'\forall\boldsymbol{\beta}\tau}^{S'\Delta} = \overline{\forall\boldsymbol{\beta}\, S'\tau}^{S'\Delta} \succ \tau'.$$

   So there is T_i' with $dom(T_i') \subseteq \boldsymbol{\beta} \cup (FV(S'\tau) - FV(S'\Delta))$ such that $\tau' = T_i'S'\tau$. Let U be the restriction of S' to $FV(\Delta, \Gamma, \boldsymbol{\alpha}, \boldsymbol{\beta})$, extended by

$$U\alpha' := T_i'S'\alpha = T_i'U\alpha \quad \text{and} \quad U\beta' := T_i'S'\beta = T_i'\beta.$$

   Then $S' =_{\Delta,\Gamma} U$ and

$$\tau' = T_i'S'\tau = \tau[T_i'S'\boldsymbol{\alpha}/\boldsymbol{\alpha}, T_i'\boldsymbol{\beta}/\boldsymbol{\beta}, S'\boldsymbol{\delta}/\boldsymbol{\delta}] = U(\tau[\boldsymbol{\alpha}'/\boldsymbol{\alpha}, \boldsymbol{\beta}'/\boldsymbol{\beta}]).$$

   U is a semiunifier of L, since $T_i'U\boldsymbol{\alpha} = U\boldsymbol{\alpha}'$, and also $T_i'U\boldsymbol{\delta} = U\boldsymbol{\delta}$, because T_i' does not operate on $FV(S'\Delta)$.

[8] Recall the convention that substitutions do not affect bound quantifiers.

(c) $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ x)$ terminates with a value different from *fail*, since we assume there is a typing assumption for x in Δ, Γ.

Case $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ (\mathtt{fn}\, x \mathtt{=>} e)) = (L, S, S\alpha \to \tau)$,

if $(L_\alpha, S, \tau) = \mathcal{W}^+_{exp}(\Delta, x : \alpha, \Gamma,\ e)$, where α is a fresh variable, and
$L := (L_\alpha - \{\, \tau' \sqsubseteq_i \tau' \mid \tau' \sqsubseteq_i \tau' \in L^\Gamma_\alpha \,\})$
$\cup\ \{\gamma \sqsubseteq_i \gamma \mid \tau' \sqsubseteq_i \tau' \in L^\Gamma_\alpha,\ \gamma \in FV(\tau') - (FV(S\alpha) - FV(S\Delta))\}$.

0. (i) By induction, S does not affect $FV(\Gamma) - FV(\Delta, x : \alpha)$. Since α does not occur in Γ, this implies the claim.

   (ii) By induction, $FV(rhs(L_\alpha), \tau) \cap FV(S\Gamma) \subseteq FV(S\Delta, x : S\alpha)$. Since $FV(rhs(L)) \subseteq FV(rhs(L_\alpha))$, it remains to show

   $$FV(S\alpha) \cap FV(S\Gamma) \subseteq FV(S\Delta). \tag{13}$$

   Clearly, if $\gamma \in FV(\Gamma) \cap FV(\Delta)$, then $FV(S\alpha) \cap FV(S\gamma) \subseteq FV(S\Delta)$. So suppose $\beta \in FV(S\alpha) \cap FV(S\gamma)$ for some $\gamma \in FV(\Gamma) - FV(\Delta)$. By 0.(i), S does not affect $\Gamma - \Delta$, hence $\gamma = S\gamma = \beta \in FV(S\alpha)$. But since $\alpha \neq \gamma$, this means that γ is affected, a contradiction. Thus, $FV(S\alpha) \cap FV(S\gamma) = \emptyset$.

   (iii) By induction, $PV(L_\alpha) \cap FV(S\Delta, S\alpha, \tau) = \emptyset$, and since $PV(L) \subseteq PV(L_\alpha)$, this gives $PV(L) \cap FV(S\Delta, S\alpha \to \tau) = \emptyset$.

   (iv) Suppose $\sqsubseteq_i$ occurs in L and $\delta \in FV(S\Delta)$. Then $\sqsubseteq_i$ occurs in L_α, and since $L_\alpha \Vdash S\Delta, x : S\alpha \ \sqsubseteq_i\ S\Delta, x : S\alpha$, there is τ' with $\tau' \sqsubseteq_i \tau' \in L_\alpha$ and $\delta \in FV(\tau')$. By definition of L we have: if $i \notin I(y)$ for all $y : \overline{\sigma} \in \Gamma$, then $\tau' \sqsubseteq_i \tau' \in L$; otherwise, $\delta \sqsubseteq_i \delta \in L$, since $\delta \in FV(S\Delta)$.

   (v) With the induction hypothesis, we have $PV(L^y) = PV(L^y_\alpha) \subseteq FV(S\overline{\sigma})$.

   (vi) Let $y : \overline{\sigma} \in \Gamma$. The claim follows by (13) from the induction hypothesis,

   $$FV(L_\alpha - L^y_\alpha) \cap FV(S\overline{\sigma}) \subseteq FV(S\Delta, x : S\alpha),$$

   if we can show that $FV(L - L^y) \subseteq FV(L_\alpha - L^y_\alpha)$. But if $\sigma_1 \sqsubseteq_i \sigma_2 \in L - L^y$ is not in $L_\alpha - L^y_\alpha$, it is of the form $\gamma \sqsubseteq_i \gamma$ with $i \in I(z)$ for some $z \neq y$, and in this case there is $\tau' \sqsubseteq_i \tau' \in L_\alpha - L^y_\alpha$ with $\gamma \in FV(\tau')$.

1. By induction, Id is a semiunifier of L_α. By the definition of L, every solution of L_α is a solution of L.

2. (ii) $\Rightarrow$ (i): Let U be a semiunifier of L with $S' =_{\Delta,\Gamma} US$ and $\sigma' = U(S\alpha \to \tau)$. To show (i), we require

   $$S'\Delta, \overline{S'\Gamma}^{S'\Delta} \ \vdash (\mathtt{fn}\, x \mathtt{=>} e) : (US\alpha \to U\tau) = \sigma'. \tag{14}$$

   Below we will modify U to obtain[9] a semiunifier $\tilde{U}$ of L_α with

   $$\tilde{U} =_{S\Delta, x:(S\alpha \to \tau)} U \quad \text{and} \quad \overline{\tilde{U}S\Gamma}^{\tilde{U}S\Delta} = \overline{US\Gamma}^{US\Delta}. \tag{15}$$

[9] Suppose $y \in FV(e)$ where $y : \beta \in \Gamma$ and $L = \{\beta \sqsubseteq_i \delta \to \delta\}$ with $i \in I(y)$, $FV(S\alpha) = \{\delta\}$ and $\Delta = \emptyset$. Then $U = [\delta/\beta, \delta/\delta]$ is a semiunifier of L, but not of $L_\alpha = L \cup \{\delta \sqsubseteq_i \delta\}$.

Since $\tilde{U}$ is a semiunifier of L_α, the induction hypothesis gives

$$\tilde{U}S\Delta,\, x : \tilde{U}S\alpha, \overline{\tilde{U}S\Gamma}^{\tilde{U}S\Delta, x:\tilde{U}S\alpha} \;\vdash e : \tilde{U}\tau.$$

By adding some quantifiers to strengthen $\overline{\tilde{U}S\Gamma}^{\tilde{U}S\Delta, x:\tilde{U}S\alpha}$ to $\overline{\tilde{U}S\Gamma}^{\tilde{U}S\Delta}$, and then using (15) to replace $\tilde{U}$ by U, we have

$$US\Delta,\, x : US\alpha, \overline{USF}^{US\Delta} \;\vdash e : U\tau. \tag{16}$$

Since $S' =_{\Delta,\Gamma} US$, we get

$$S'\Delta, x : US\alpha, \overline{S'\Gamma}^{S'\Delta} \;\vdash e : U\tau,$$

which gives (14) by an application of the typing rule for abstractions.

It remains to show the existence of a semiunifier $\tilde{U}$ of L_α satisfying (15). By assumption, U can be expanded to a solution $(U, T_1, \ldots, T_n)$ of L. If it is not a solution of L_α, there is $\tau' \sqsubseteq_i \tau' \in L_\alpha^\Gamma$ and some $\delta \in FV(\tau') \cap FV(S\alpha) - FV(S\Delta)$ such that $T_i U\delta \neq U\delta$. Hence for some $\gamma \in FV(U\delta)$, we have $T_i\gamma \neq \gamma$, which implies[10] $\gamma \in PV(U\sigma \sqsubseteq_i U\rho)$ for some inequation $\sigma \sqsubseteq_i \rho \in L^\Gamma$. So γ is a witness that $(U, T_1, \ldots, T_n)$ does not solve

$$\{\delta \sqsubseteq_i \delta,\, \sigma \sqsubseteq_i \rho\} \quad \text{resp.} \quad \{\tau' \sqsubseteq_i \tau',\, \sigma \sqsubseteq_i \rho\} \subseteq L_\alpha^\Gamma \subseteq L_\alpha\,.$$

Modify $(U, T_1, \ldots, T_n)$ as follows. For each i with $\sqsubseteq_i$ occuring in L^Γ, let

$$D_i := \{\, \delta \in FV(S\alpha) - FV(S\Delta) \mid \delta \in FV(\tau'),\, \tau' \sqsubseteq_i \tau' \in L_\alpha^\Gamma \text{ for some } \tau' \,\},$$

and let $PV_i(UL)$ be the pattern variables of UL with respect to $\sqsubseteq_i$. Let

$$W := \bigcup \{\, W_i \mid \sqsubseteq_i \text{ occurs in } L^\Gamma \,\} \quad \text{with} \quad W_i := FV(UD_i) \cap PV_i(UL^\Gamma),$$

be the set of variables witnessing that $(U, T_1, \ldots, T_n)$ does not solve L_α. Since U must not be changed on D_i, we can only modify U on L^Γ to obtain a suitable $\tilde{U}$ with $PV_i(\tilde{U}L^\Gamma) \cap FV(\tilde{U}D_i) = \emptyset$. For each $\gamma \in W$, let $\tilde{\gamma}$ be a fresh variable, and with these put

$$S\gamma := \begin{cases} \tilde{\gamma}, & \text{if } \gamma \in W, \\ \gamma, & \text{otherwise.} \end{cases}$$

Using $W_i := \emptyset$ if $\sqsubseteq_i$ does not occur in L^Γ, define $(\tilde{U}, \tilde{T}_1, \ldots, \tilde{T}_n)$ by

$$\tilde{U}\beta = \begin{cases} U\beta, & \text{if } \beta \notin PV(L^\Gamma), \\ SU\beta, & \text{else,} \end{cases}$$

$$\tilde{T}_i\gamma' = \begin{cases} T_i\gamma, & \text{if } \gamma' \equiv S\gamma \text{ for some } \gamma \in W, \\ \gamma' & \text{if } \gamma' \in W_i, \\ T_i\gamma', & \text{else.} \end{cases}$$

Since, by induction hypothesis 0.(iii), $PV(L_\alpha) \cap FV(S\Delta, S\alpha, \tau) = \emptyset$, we have

$$\tilde{U} =_{S\Delta, x:(S\alpha \to \tau)} U. \tag{17}$$

[10] else we could change T_i on γ

To show that $(\tilde{U}, \tilde{T}_1, \ldots, \tilde{T}_n)$ solves L_α, pick $\sigma \sqsubseteq_i \rho \in L_\alpha$. By induction hypothesis 0.(v),

$$FV(\rho) \cap PV(L^\Gamma) \subseteq FV(rhs(L_\alpha)) \cap PV(L_\alpha^\Gamma) = \emptyset$$

and hence $\tilde{U}\rho = U\rho$ by definition. If $\sigma \sqsubseteq_i \rho \in L_\alpha - L_\alpha^\Gamma$, then, by induction hypothesis 0.(v), $PV(L^\Gamma) \cap FV(\sigma) \subseteq PV(L_\alpha^\Gamma) \cap FV(\sigma) = \emptyset$, so

$$\tilde{U}\sigma = U\sigma,\ SW \cap FV(U\sigma) = \emptyset = W_i \cap FV(U\sigma),$$

and thus $\tilde{T}_i\tilde{U}\sigma = \tilde{T}_iU\sigma = T_iU\sigma$. Since $(U, T_1, \ldots, T_n)$ solves $L \supseteq L_\alpha - L_\alpha^\Gamma$, we also have $T_iU\sigma = U\rho = \tilde{U}\rho$, which shows that $(\tilde{U}, \tilde{T}_i)$ solves $\sigma \sqsubseteq_i \rho$.

If $\sigma \sqsubseteq_i \rho \in L_\alpha^\Gamma \cap L^\Gamma$, again we have $T_iU\sigma = U\rho$, and it remains to show

$$\tilde{T}_i\tilde{U}\beta = T_iU\beta \quad \text{for each } \beta \in FV(\sigma).$$

If $\beta \notin PV_i(L)$, then since $\sigma \sqsubseteq_i \rho \in L_\alpha$ is semiunified by Id, according to claim 1., at the same positions where β occurs in σ it also occurs in ρ. This implies $\tilde{U}\beta = U\beta$, so $SW \cap FV(\tilde{U}\beta) = \emptyset$, and $T_iU\beta = U\beta$, so $FV(U\beta) \cap PV_i(UL^\Gamma) = \emptyset$ and $FV(U\beta) \cap W_i = \emptyset$. Therefore, $\tilde{T}_i\tilde{U}\beta = \tilde{T}_iU\beta = T_iU\beta$.
If $\beta \in PV_i(L)$, then $\tilde{T}_i\tilde{U}\beta = \tilde{T}_iSU\beta = T_iU\beta$ since $W_i \cap FV(SU\beta) = \emptyset$.

Finally, if $\sigma \sqsubseteq_i \rho \in L_\alpha^\Gamma - L^\Gamma$, it has the form $\tau' \sqsubseteq_i \tau'$, and $\tilde{U}\tau' = U\tau'$ contains none of the fresh variables of SW. Suppose $\delta \in FV(\tau')$. If $\delta \sqsubseteq_i \delta \in L^\Gamma$, then $FV(U\delta) \cap W_i \subseteq FV(U\delta) \cap PV_i(UL^\Gamma) = \emptyset$, whence

$$\tilde{T}_i\tilde{U}\delta = \tilde{T}_iU\delta = T_iU\delta = U\delta = \tilde{U}\delta.$$

If $\delta \sqsubseteq_i \delta \notin L^\Gamma$, then $\delta \in FV(S\alpha) - FV(S\Delta)$ and so $\delta \in D_i$. We check $\tilde{T}_i\tilde{U}\delta = \tilde{U}\delta$ $(= U\delta)$. For $\gamma' \in FV(U\delta) \cap W_i$ we have $\tilde{T}_i\gamma' = \gamma'$ $(\neq T_i\gamma'!)$, and for $\gamma' \in FV(U\delta) - W_i$ we have $\tilde{T}_i\gamma' = T_i\gamma' = \gamma'$, since γ' was not in $PV_i(UL^\Gamma)$. This showed $\tilde{T}_i\tilde{U}\tau' = U\tau' = \tilde{U}\tau'$.

Since $(\tilde{U}, \tilde{T}_1, \ldots, \tilde{T}_n)$ solves L_α, we now know that $\tilde{U}$ is a semiunifer of L_α.
To finish the proof of (15), we show

$$\overline{\tilde{U}S\Gamma}^{\tilde{U}S\Delta} = \overline{US\Gamma}^{US\Delta}.$$

Since $\tilde{U} =_{S\Delta} U$, on both sides the type variables universally quantified are those not in $US\Delta$. We only have to care about $\beta \in FV(S\Gamma) - FV(S\Delta, S\alpha, \tau)$ with $\tilde{U}\beta \neq U\beta$. In this case, $\beta \in PV(L^\Gamma)$, and there is $\gamma \in FV(U\beta) - FV(\tilde{U}\beta)$ that has been renamed to a fresh $\tilde{\gamma}$ in $\tilde{U}\beta$, whence for some i,

$$\gamma \in FV(UD_i) \cap PV_i(UL^\Gamma).$$

Then $\sqsubseteq_i$ is in L^Γ, which, by induction hypothesis 0.(iv), contains some $\tau'(\delta, \ldots) \sqsubseteq_i \tau'(\delta, \ldots)$ for each $\delta \in FV(S\Delta)$. Hence $FV(US\Delta) \cap PV_i(UL^\Gamma) = \emptyset$ and so $\gamma \notin FV(US\Delta)$. It follows that fresh variables $\tilde{\gamma}$ of $\tilde{U}S\Gamma$ occur in $\tilde{U}\beta$ *for* $\beta \in FV(S\Gamma) - FV(S\Delta)$ *only*, and hence

$$\overline{\tilde{U}S\Gamma}^{US\Delta} = (\ldots \forall\tilde{\gamma} \ldots)\ \tilde{U}S\Gamma \quad = \quad (\ldots \forall\gamma \ldots)\ US\Gamma = \overline{US\Gamma}^{US\Delta}$$

are equal up to renaming of bound variables.

(i) $\Rightarrow$ (ii): Suppose for some (S', τ'), we have $S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash (\mathtt{fn}\, x \mathtt{=>} e) : \tau'$. By the typing rule for function expressions, there are monotypes τ_1, τ_2 with

$$S'\Delta,\, x : \tau_1,\, \overline{S'\Gamma}^{S'\Delta} \vdash e : \tau_2 \quad \text{and} \quad \tau' = \tau_1 \to \tau_2.$$

We may assume that $\overline{S'\Gamma}^{S'\Delta} = \overline{S'\Gamma}^{S'\Delta, x:\tau_1}$ up to renaming of bound variables, because $FV(S'\Gamma) - FV(S'\Delta)$ could be renamed to make it disjoint from $FV(\tau_1) - FV(S'\Delta)$. Since α was fresh, we may also assume $S'\alpha = \tau_1$, so that

$$S'\Delta,\, x : S'\alpha,\, \overline{S'\Gamma}^{S'\Delta,\, x:S'\alpha} \vdash e : \tau_2.$$

By induction, there is a semiunifier U of L_α, hence of L, with $S' =_{\Delta, x:\alpha, \Gamma} US$ and $\tau_2 = U\tau$. So U is a semiunifier of L with

$$S' =_{\Delta,\Gamma} US \quad \text{and} \quad \tau' = U(S\alpha \to \tau).$$

(c) Suppose $(\mathtt{fn}\, x \mathtt{=>} e)$ is ML^+-typable modulo Δ, Γ. Then there are (S, τ) such that

$$S\Delta, \overline{S\Gamma}^{S\Delta} \vdash (\mathtt{fn}\, x \mathtt{=>} e) : \tau.$$

By the typing rule for function expressions, $\tau = \tau_1 \to \tau_2$ and

$$S\Delta, x : \tau_1, \overline{S\Gamma}^{S\Delta} \vdash e : \tau_2.$$

Since by renaming of bound variables, we may assume $FV(\tau) \cap FV(S\Gamma) \subseteq FV(S\Delta)$, we have

$$S\Delta, x : \tau_1, \overline{S\Gamma}^{S\Delta, x:\tau_1} \vdash e : \tau_2\,.$$

Hence for fresh α, e is typable modulo $\Delta, x : \alpha, \Gamma$. By induction, $\mathcal{W}^+_{exp}(\Delta, x : \alpha, \Gamma,\ e)$ is a triple $(L_\alpha, S_\alpha, \tau_\alpha)$. By the definition of $\mathcal{W}^+_{exp}$, it follows that $\mathcal{W}^+_{exp}(\Delta, \Gamma\ (\mathtt{fn}\, x \mathtt{=>} e))$ returns $(L^-_\alpha, S_\alpha, S_\alpha\alpha \to \tau_\alpha)$ for some $L^-_\alpha \subseteq L_\alpha$.

Case $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ e_1 \cdot e_2) = (US_2L_1 \cup UL_2,\ US_2S_1,\ U\alpha)$

if $(L_1, S_1, \tau_1) = \mathcal{W}^+_{exp}(\Delta, \Gamma, e_1)$
$(L_2, S_2, \tau_2) = \mathcal{W}^+_{exp}(S_1\Delta, S_1\Gamma, e_2)$
$U = mgu(S_2\tau_1, \tau_2 \to \alpha)$, where α is a fresh variable.

0. (i) By induction, S_1 does not affect variables of $FV(\Gamma) - FV(\Delta)$ and S_2 does not affect $FV(S_1\Gamma) - FV(S_1\Delta)$. By Proposition 13 (i), S_2S_1 does not affect $FV(\Gamma) - FV(\Delta)$. To show that US_2S_1 does not affect $FV(\Gamma) - FV(\Delta)$, it remains to show that U does not affect $FV(S_2S_1\Gamma) - FV(S_2S_1\Delta)$. Note that U affects only variables in $FV(S_2\tau_1, \tau_2, \alpha)$ and we need not consider $\alpha \notin FV(S_2S_1\Gamma) - FV(S_2S_1\Delta)$. By induction hypothesis 0.(ii),

$$FV(\tau_1) \cap FV(S_1\Gamma) \subseteq FV(S_1\Delta), \quad FV(\tau_2) \cap FV(S_2S_1\Gamma) \subseteq FV(S_2S_1\Delta).$$

Since S_2 does not affect $FV(S_1\Gamma) - FV(S_1\Delta)$, by Proposition 13 (ii) we have

$$FV(S_2\tau_1) \cap FV(S_2S_1\Gamma) \subseteq FV(S_2S_1\Delta).$$

Hence, variables of $FV(S_2S_1\Gamma) - FV(S_2S_1\Delta)$ are not in $FV(S_2\tau_1, \tau_2)$ and thus not affected by U.

(ii) From the induction hypothesis for e_1 we have

$$FV(rhs(L_1), \tau_1) \cap FV(S_1\Gamma) \subseteq FV(S_1\Delta),$$

from which, using hypothesis 0.(i) for e_2, Proposition 13 (ii) gives

$$FV(rhs(S_2L_1), S_2\tau_1) \cap FV(S_2S_1\Gamma) \subseteq FV(S_2S_1\Delta).$$

By the induction hypothesis for e_2, we have

$$FV(rhs(L_2)) \cap FV(S_2S_1\Gamma) \subseteq FV(S_2S_1\Delta).$$

Since U does not affect $FV(S_2S_1\Gamma) - FV(S_2S_1\Delta)$ we therefore have

$$FV(rhs(US_2L_1 \cup UL_2), US_2\tau_1) \cap FV(US_2S_1\Gamma) \subseteq FV(US_2S_1\Delta),$$

by Proposition 13 (ii). The claim follows since $U\alpha$ is a subterm of $US_2\tau_1$.

(iii) By the induction hypothesis for e_1,

$$PV(L_1) \cap FV(S_1\Delta, \tau_1) = \emptyset. \tag{18}$$

By the construction of S_2 we may assume that all variables of L_1 that are affected by S_2 belong to $FV(S_1\Delta, S_1\Gamma)$, and by induction hypothesis 0.(i) for e_2, they must belong to $FV(S_1\Delta)$. Because of (18) then, S_2 does not affect $PV(L_1)$ at all. By Lemma 12 and induction hypothesis 1 for e_1, S_2L_1 holds and $PV(S_2L_1) = PV(L_1)$. Hence, since S_2 does not affect $PV(L_1)$,

$$PV(S_2L_1) \cap FV(S_2S_1\Delta, S_2\tau_1) = \emptyset, \tag{19}$$

using (18) again. By the induction hypothesis for e_2,

$$PV(L_2) \cap FV(S_2S_1\Delta, \tau_2) = \emptyset. \tag{20}$$

Since L_2 holds by induction hypothesis 1. for e_2, and S_2L_1 and L_2 have no inequation relations in common, $L := S_2L_1 \cup L_2$ holds. We claim that U does not affect $PV(L)$. By our choice of mgu, $U = mgu(S_2\tau_1, \tau_2 \rightarrow \alpha)$ only affects variables in $FV(S_2\tau_1, \tau_2, \alpha)$. By (19), (20) and since α was fresh, it is sufficient to show

$$FV(S_2\tau_1) \cap PV(L_2) = \emptyset = FV(\tau_2) \cap PV(S_2L_1).$$

Note that by the construction of τ_2 and S_2 from $S_1\Delta, S_1\Gamma$, we may assume

$$FV(\tau_2) \cap FV(S_2L_1) \subseteq FV(S_2S_1\Delta, S_2S_1\Gamma).$$

Since $FV(\tau_2) \cap FV(S_2 S_1 \Gamma) \subseteq FV(S_2 S_1 \Delta)$ by induction hypothesis 0.(ii) for e_2, from (19) we obtain $FV(\tau_2) \cap PV(S_2 L_1) = \emptyset$.

By the construction of L_2 and S_2, we may assume that a variable $\alpha \in FV(\tau_1) - FV(S_1\Delta, S_1\Gamma)$ does not occur in L_2 and is not affected by S_2, whence $FV(S\alpha) \cap PV(L_2) = \emptyset$. For $\alpha \in FV(\tau_1) \cap FV(S_1\Delta, S_1\Gamma)$ we have $\alpha \in FV(S_1\Delta)$ by induction hypothesis 0.(ii) for e_1, and then $FV(S\alpha) \cap PV(L_2) = \emptyset$ by (20). Hence $FV(S_2\tau_1) \cap PV(L_2) = \emptyset$.

Having shown that U does not affect $PV(L)$, by Lemma 12 we conclude that UL holds, which proves claim 1, and that $PV(UL) = PV(L)$. Since U does not affect $PV(L)$, from (19), (20) and $\alpha \notin FV(L)$ we get the claim,

$$PV(UL) \cap FV(US_2S_1\Delta, U\alpha) = \emptyset.$$

(iv) Let $L := S_2L_1 \cup L_2$ again and $\sqsubseteq$ occur in UL and $\delta \in FV(US_2S_1\Delta)$. Then $\sqsubseteq$ occurs either in L_1 or in L_2. Suppose $\sqsubseteq$ occurs in L_1. There is $\delta_1 \in FV(S_1\Delta)$ such that $\delta \in FV(US_2\delta_1)$. Since, by induction, $L_1 \vDash S_1\Delta \sqsubseteq S_1\Delta$, there is τ_1 with $\delta_1 \in FV(\tau_1)$ and $\tau_1 \sqsubseteq \tau_1 \in L_1$. Hence for $\tau' := US_2\tau_1$ we have $\delta \in FV(\tau')$ and $\tau' \sqsubseteq \tau' \in US_2L_1$. The case when $\sqsubseteq$ occurs in L_2 is similar. This shows that $UL \vDash US_2S_1\Delta \sqsubseteq US_2S_1\Delta$.

(v) Let $y : \overline{\sigma} \in \Gamma$. We have $PV(L_1^y) \subseteq FV(S_1\overline{\sigma})$ by the induction hypothesis for e_1, and since S_2 does not affect $PV(L_1)$, this implies $PV(S_2L_1^y) \subseteq FV(S_2S_1\overline{\sigma})$. Since $PV(L_2^y) \subseteq FV(S_2S_1\overline{\sigma})$ by the induction hypothesis for e_2, for $L := S_2L_1 \cup L_2$ and $S := S_2S_1$ we get

$$PV(L^y) = PV(S_2L_1^y) \cup PV(L_2^y) \subseteq FV(S\overline{\sigma}).$$

Since U does not affect the pattern variables of L, the claim follows by Proposition 13.

(vi) Again, let $y : \overline{\sigma} \in \Gamma$, $L := S_2L_1 \cup L_2$ and $S := S_2S_1$. Since U does not affect $F(S\Gamma) - FV(S\Delta)$, it is sufficient to show

$$FV(L - L^y) \cap FV(S\overline{\sigma}) \subseteq FV(S\Delta).$$

Note that since L_1 and L_2 have no inequation relation in common,

$$L - L^y = (S_2L_1 - S_2L_1^y) \cup (L_2 - L_2^y),$$

so we show the claim for each summand. By the induction hypothesis for e_1,

$$FV(L_1 - L_1^y) \cap FV(S_1\overline{\sigma}) \subseteq FV(S_1\Delta).$$

Since, by hypothesis 0.(i) for e_2, S_2 does not affect $FV(S_1\overline{\sigma}) - FV(S_1\Delta)$,

$$FV(S_2L_1 - S_2L_1^y) \cap FV(S\overline{\sigma}) \subseteq FV(S\Delta).$$

By the induction hypothesis for e_2, we also have

$$FV(L_2 - L_2^y) \cap FV(S\overline{\sigma}) \subseteq FV(S\Delta).$$

1. This has been shown in the proof of 0.(iii).
2. By induction on the two calls of $\mathcal{W}^+_{exp}$, we have:

   e_1) For each (S', τ_1'), the following are equivalent:

   (i) $S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash e_1 : \tau_1'$,

   (ii) for some semiunifier U_1 of L_1, $S' =_{\Delta,\Gamma} U_1 S_1$ and $\tau_1' = U_1\tau_1$.

   e_1) For each (S', τ_2'), the following are equivalent:

   (i) $S'S_1\Delta, \overline{S'S_1\Gamma}^{S'S_1\Delta} \vdash e_2 : \tau_2'$

   (ii) for some semiunifier U_2 of L_2, $S' =_{S_1\Delta, S_1\Gamma} U_2 S_2$ and $\tau_2' = U_2\tau_2$.

   (i) $\Rightarrow$ (ii): Suppose for some (S', σ), that $S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash e_1 e_2 : \sigma$. By the typing rule for application expressions, for some τ_2' we have

   $$S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash e_1 : \tau_2' \to \sigma \quad \text{and} \quad S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash e_2 : \tau_2'.$$

   By the induction hypothesis for e_1 there is a semiunifier U_1 of L_1 with

   $$S' =_{\Delta,\Gamma} U_1 S_1 \quad \text{and} \quad \tau_2' \to \sigma = U_1\tau_1$$

   and by the one for e_2 there is a semiunifier U_2 of L_2 with

   $$U_1 =_{S_1\Delta, S_1\Gamma} U_2 S_2 \quad \text{and} \quad \tau_2' = U_2\tau_2.$$

   Since we may assume that $FV(L_1) \cap FV(L_2) \subseteq FV(S_1\Delta, S_1\Gamma)$, and the typing of e_2 does not envolve $FV(\tau_1) - FV(S_1\Delta)$, we can extend U_2S_2 to the variables of L_1 and τ_1 such that

   $$U_1 =_{L_1,\tau_1} U_2 S_2.$$

   Hence $U_2S_2\tau_1 = U_1\tau_1 = \tau_2' \to \sigma = U_2\tau_2 \to \sigma$, and since α was fresh, by putting $U_2\alpha = \sigma$ we can modify U_2 to a unifier of $S_2\tau_1$ and $\tau_2 \to \alpha$. Hence for some $\tilde{U}$, we have $U_2 =_{S_2\tau_1, \tau_2, \alpha} \tilde{U}U$, and since $U = mgu(S_2\tau_1, \tau_2 \to \alpha)$ only affects variables in $FV(S_2\tau_1, \tau_2 \to \alpha)$, we can assume $U_2\beta = \tilde{U}U\beta$ for *all* variables. Therefore,

   $$S' =_{\Delta,\Gamma} U_1S_1 =_{\Delta,\Gamma} U_2S_2S_1 =_{\Delta,\Gamma} \tilde{U}US_2S_1 \quad \text{and} \quad \tilde{U}U\alpha = U_2\alpha = \sigma.$$

   It remains to be shown that $\tilde{U}$ is a semiunifier of $US_2L_1 \cup UL_2$. As we saw,

   $$\tilde{U}US_2L_1 = U_2S_2L_1 = U_1L_1 \quad \text{and} \quad \tilde{U}UL_2 = U_2L_2$$

   hold, so $\tilde{U}$ is a semiunifier of both US_2L_1 and UL_2. Since L_1 and L_2 have no inequation relations in common, $\tilde{U}$ is a semiunifier of their union.

   (ii) $\Rightarrow$ (i): Suppose $\tilde{U}$ is a semiunifier of $US_2L_1 \cup UL_2$, $\tau_2 = \tilde{U}U\alpha$ and $S' =_{\Delta,\Gamma} \tilde{U}US_2S_1$. Note that $U_2 := \tilde{U}U$ is a semiunifier of L_2 and $U_1 := U_2S_2 = \tilde{U}US_2$ is a semiunifier of L_1. Hence by the induction hypotheses for e_1 and e_2 we have

   $$U_2S_2S_1\Delta, \overline{U_2S_2S_1\Gamma}^{U_2S_2S_1\Delta} \vdash e_2 : U_2\tau_2, \quad U_1S_1\Delta, \overline{U_1S_1\Gamma}^{U_1S_1\Delta} \vdash e_1 : U_1\tau_1.$$

   Since $U_1\tau_1 = \tilde{U}US_2\tau_1 = \tilde{U}(U\tau_2 \to U\alpha) = U_2\tau_2 \to U_2\alpha$, the typing rule for application expressions can be applied and gives

   $$\tilde{U}US_2S_1\Delta, \overline{\tilde{U}US_2S_1\Gamma}^{\tilde{U}US_2S_1\Delta} \vdash e_1e_2 : \tilde{U}U\alpha,$$

   which establishes the claim.

3. This is proved similar to the case for `let` d `in` e `end`.

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

0. (i) By induction, S_1 does not affect $FV(\Gamma) - FV(\Delta)$ and S_2 does not affect $FV(S_1\Gamma, x : \overline{\tau}) - FV(S_1\Delta)$. Proposition 13 gives the claim.

   (ii) By induction on d, $FV(rhs(L_1), \overline{\tau}) \cap FV(S_1\Gamma) \subseteq FV(S_1\Delta)$. Note that, in particular, for each $y : \overline{\sigma} \in \Gamma$ we have

   $$FV(S_1\overline{\sigma}) \cap FV(\overline{\tau}) \subseteq FV(S_1\Delta),$$

   so that –together with 0.(i)– the extended environment $S_1\Delta, S_1\Gamma, x : \overline{\tau}$ fulfills the assumption for applying $\mathcal{W}^+_{exp}$ to e. Hence, the induction hypothesis for e gives $FV(rhs(L_2), \rho) \cap FV(S_2S_1\Gamma) \subseteq FV(S_2S_1\Delta)$. Using Proposition 13, we can combine the induction hypotheses to obtain

   $$FV(rhs(S_2L_1 \cup L_2), \rho) \cap FV(S_2S_1\Gamma) \subseteq FV(S_2S_1\Delta).$$

   (iii) We first show that S_2 does not affect $PV(L_1)$. All variables of L_1 affected by S_2 belong to $FV(S_1\Delta, S_1\Gamma, \overline{\tau})$. By hypothesis 0.(i) for e, S_2 does not affect $FV(S_1\Gamma, x : \overline{\tau}) - FV(S_1\Delta)$, and by hypothesis 0.(iii) for d,

   $$PV(L_1) \cap FV(S_1\Delta) = \emptyset,$$

   so S_2 does not affect $PV(L_1)$ at all. We can now prove the claim

   $$PV(S_2L_1 \cup L_2) \cap FV(S_2S_1\Delta, \rho) = \emptyset.$$

   By the induction hypothesis 0.(iii) for d, $PV(L_1) \cap FV(S_1\Delta) = \emptyset$, and since S_2 does not affect $PV(L_1)$, this gives

   $$PV(S_2L_1) \cap FV(S_2S_1\Delta) = \emptyset.$$

   For $PV(S_2L_1) \cap FV(\rho)$, note that by hypothesis 0.(ii) for e,

   $$FV(\rho) \cap FV(S_2S_1\Gamma, S_2\overline{\tau}) \subseteq FV(S_2S_1\Delta).$$

   Since $FV(\rho) \cap PV(S_2L_1) \subseteq FV(\rho) \cap FV(S_2S_1\Delta, S_2S_1\Gamma, S_2\overline{\tau})$, we get

   $$PV(S_2L_1) \cap FV(S_2S_1\Delta, \rho) = \emptyset.$$

   By the induction hypothesis for e, we also have $PV(L_2) \cap FV(S_2S_1\Delta, \rho)$, and the claim is proven.

   (iv) Use the induction hypotheses 0.(iv) for d and e, similar as for $e_1 \cdot e_2$.

(v) Suppose $y : \overline{\sigma} \in \Gamma$. Let $S := S_2 S_1$ and $L := S_2 L_1 \cup L_2$. By induction on d, $PV(L_1^y) \subseteq FV(S_1\overline{\sigma})$, and since S_2 does not affect $PV(L_1)$, this implies

$$PV(S_2 L_1^y) \subseteq FV(S_2 S_1 \overline{\sigma}).$$

On the other hand, by induction on e we have $PV(L_2^y) \subseteq FV(S_2 S_1 \overline{\sigma})$, and the claim $PV(L^y) \subseteq FV(S\overline{\sigma})$ follows.

(vi) Suppose $y : \overline{\sigma} \in \Gamma$, and let $S := S_2 S_1$ and $L := S_2 L_1 \cup L_2$. Since L_1 and L_2 have no inequation relation in common,

$$L - L^y = (S_2 L_1 - S_2 L_1^y) \cup (L_2 - L_2^y).$$

By the induction hypotheses for d,

$$FV(L_1 - L_1^y) \cap FV(S_1\overline{\sigma}) \subseteq FV(S_1\Delta),$$

and since, by hypothesis 0.(i) for e, S_2 does not affect $FV(S_1\Gamma) - FV(S_1\Delta)$,

$$FV(S_2 L_1 - S_2 L_1^y) \cap FV(S\overline{\sigma}) \subseteq FV(S\Delta).$$

By the induction hypothesis for e, we also have

$$FV(L_2 - L_2^y) \cap FV(S\overline{\sigma}) \subseteq FV(S\Delta).$$

The claim $FV(L - L^y) \cap FV(S\overline{\sigma}) \subseteq FV(S\Delta)$ is shown.

1. By induction, L_1 and L_2 hold. Since –as shown above– S_2 does not affect $PV(L_1)$, by Lemma 12, $S_2 L_1$ holds. Since L_2 and $S_2 L_1$ have no inequation relations in common, $S_2 L_1 \cup L_2$ holds.

2. Let $\overline{\tau} = \forall \vec{\beta} \tau$. From the recursive calls to $\mathcal{W}^+$, by induction we have

   d) For each $(S', \{x : \overline{\sigma}\})$, the following are equivalent:

      (i) $(S', \{x : \overline{\sigma}\})$ is a typing of d modulo Δ, Γ

      (ii) for some semiunifier U_1 of L_1, $S' =_{\Delta,\Gamma} U_1 S_1$ and $\overline{\sigma} = \overline{U_1\tau}^{U_1 S_1 \Delta}$

   e) For all (S', ρ'), the following are equivalent:

      (i) (S', ρ') is a typing of e modulo $S_1\Delta, S_1\Gamma, x : \overline{\tau}$

      (ii) for some semiunifier U_2 of L_2, $S' =_{S_1\Delta, S_1\Gamma, x:\overline{\tau}} U_2 S_2$ and $\rho' = U_2\rho$.

(i) $\Rightarrow$ (ii): Suppose (S', ρ') is a typing of $\texttt{let}\, d\, \texttt{in}\, e\, \texttt{end}$ modulo Δ, Γ. According to the typing rule for `let`-expressions, for some $x : \overline{\sigma}$ we have

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta} \mid\!- d : \{x : \overline{\sigma}\} \quad \text{and} \quad S'\Delta, \overline{S'\Gamma}^{S'\Delta}, x : \overline{\sigma} \mid\!- e : \rho'.$$

By the induction hypothesis for d, there is a semiunifier U_1 of L_1 with

$$S' =_{\Delta,\Gamma} U_1 S_1 \quad \text{and} \quad \overline{\sigma} = \overline{U_1\tau}^{U_1 S_1 \Delta}.$$

The typing for e can hence be written as

$$U_1S_1\Delta, \overline{U_1S_1\Gamma}^{U_1S_1\Delta}, x : \overline{U_1\tau}^{U_1S_1\Delta} \vdash e : \rho'.$$

Of course we can strengthen the assumption $x : \overline{U_1\tau}^{U_1S_1\Delta}$ to $x : \overline{U_1\overline{\tau}}^{U_1S_1\Delta}$. Then, by the induction hypothesis for e, there is a semiunifier U_2 of L_2 with

$$U_1 =_{S_1\Delta, S_1\Gamma, x:\overline{\tau}} U_2S_2 \quad \text{and} \quad \rho' = U_2\rho.$$

It follows that $S' =_{\Delta,\Gamma} U_1S_1 =_{\Delta,\Gamma} U_2S_2S_1$. Because $FV(L_1) \cap FV(L_2) \subseteq FV(S_1\Delta, S_1\Gamma, \overline{\tau})$, we can modify U_2 on $FV(S_2L_1) - FV(S_2S_1\Delta, S_2S_1\Gamma S_2\overline{\tau})$ so that $U_2S_2L_1 = U_1L_1$. Then U_2 is a semiunifier of both L_2 and S_2L_1, and since these have no ineqaution relations in common, U_2 is a semiunifier of $S_2L_1 \cup L_2$.

(ii) $\Rightarrow$ (i): Let U_2 be a semiunifier of $S_2L_1 \cup L_2$ with $S' =_{\Delta,\Gamma} U_2S_2S_1$ and $\rho' = U_2\rho$. Then $U_1 := U_2S_2$ is a semiunifier of L_1, and by the induction hypothesis for d we have

$$U_1S_1\Delta, \overline{U_1S_1\Gamma}^{U_1S_1\Delta} \vdash d : \{x : \overline{U_1\tau}^{U_1S_1\Delta}\}.$$

By the induction hypothesis for d also, $FV(\overline{\tau}) \subseteq FV(S_1\Delta) \cup spec(S_1\Gamma, L_1)$. If we modify the semiunifier U_1 as in Lemma 15, we get the stronger typing

$$U_1S_1\Delta, \overline{U_1S_1\Gamma}^{U_1S_1\Delta} \vdash d : \{x : \overline{U_1\overline{\tau}}^{U_1S_1\Delta}\}. \tag{21}$$

Since U_2 is also a semiunifier of L_2, by the induction hypothesis for e we obtain that $(U_2S_2, \rho') = (U_1, \rho')$ is a typing of e modulo $S_1\Delta, S_1\Gamma, x : \overline{\tau}$, i.e.

$$U_1S_1\Delta, \overline{U_1S_1\Gamma}^{U_1S_1\Delta}, x : \overline{U_1\overline{\tau}}^{U_1S_1\Delta} \vdash e : \rho'. \tag{22}$$

An application of the typing rule for `let`-expressions to (21) and (22) gives

$$U_1S_1\Delta, \overline{U_1S_1\Gamma}^{U_1S_1\Delta} \vdash \texttt{let}\ d\ \texttt{in}\ e\ \texttt{end} : \rho'.$$

Since $U_1S_1 =_{\Delta,\Gamma} S'$, this shows that (S', ρ') is a typing of $\texttt{let}\ d\ \texttt{in}\ e\ \texttt{end}$ modulo Δ, Γ.

(c) Suppose $\texttt{let}\ d\ \texttt{in}\ e\ \texttt{end}$ is typable modulo Δ, Γ. Then there are S' and ρ' such that

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash \texttt{let}\ d\ \texttt{in}\ e\ \texttt{end} : \rho'.$$

By the typing rule for `let`-expressions, there is a polytype $\overline{\sigma}$ with

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash d : \{x : \overline{\sigma}\} \quad \text{and} \quad S'\Delta, \overline{S'\Gamma}^{S'\Delta}, x : \overline{\sigma} \vdash e : \rho',$$

where $FV(\overline{\sigma}) \subseteq FV(S'\Delta, \overline{S'\Gamma}^{S'\Delta}) = FV(S'\Delta)$. By induction on d,

$$\mathcal{W}^+_{dec}(\Delta, \Gamma,\ d) = (L_1, S_1, \{x : \overline{\tau}\})$$

for some L_1, S_1 and $\overline{\tau} = \forall\boldsymbol{\beta}\,\tau$. By 2.(ii), there is a semiunifier U of L_1 such that

$$S' =_{\Delta,\Gamma} US_1 \quad \text{and} \quad \overline{\sigma} = \overline{U\tau}^{US_1\Delta}.$$

Since we now have

$$US_1\Delta, \overline{US_1\Gamma}^{US_1\Delta}, x : \overline{U\tau}^{US_1\Delta} \vdash e : \rho',$$

e is typable modulo $S_1\Delta, S_1\Gamma, x : \tau$, and since $\overline{\tau} \succ \tau$, also modulo $S_1\Delta, S_1\Gamma, x : \overline{\tau}$. By induction,

$$\mathcal{W}^+_{exp}(S_1\Delta, S_1\Gamma, x : \overline{\tau},\ e) = (L_2, S_2, \rho)$$

for suitable L_2, S_2, ρ. The claim follows by definition of $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ \texttt{let}\, d\, \texttt{in}\, e\, \texttt{end})$.

Case $\mathcal{W}^+_{dec}(\Delta, \Gamma,\ \texttt{val rec}\, f = e) = (\tilde{U}(L[\tau/\alpha]), \tilde{U}S, \{f : \forall \boldsymbol{\beta}\, \tilde{U}\tau\})$
if $(L, S, \tau) = \mathcal{W}^+_{exp}(\Delta, \Gamma, f : \alpha,\ e)$, with α fresh,
$\tilde{U} = mgsu(L[\tau/\alpha])$
$\boldsymbol{\beta} = FV(\tilde{U}\tau) - FV(\tilde{U}S\Delta) - spec(\tilde{U}S\Gamma, \tilde{U}(L[\tau/\alpha]))$

0. (i) By induction, S does not affect $FV(\Gamma, f : \alpha) - FV(\Delta)$, whence by Proposition 13 it is sufficient to show that $\tilde{U}$ does not affect $FV(S\Gamma) - FV(S\Delta)$. Since $mgsu(L[\tau/\alpha])$ does not trivially rename variables of $L[\tau/\alpha]$, by Lemma 14 it suffices to show that variables of $FV(S\Gamma) - FV(S\Delta)$ do not occur in $rhs(L[\tau/\alpha])$.

   Since α is not affected by S, it is not in $FV(S\Delta)$. Hence by induction hypothesis 0.(ii), $\alpha = S\alpha \in FV(S\Gamma, f : \alpha)$ also cannot be in $FV(rhs(L))$. So $rhs(L) = rhs(L[\tau/\alpha])$, and by induction hypothesis 0.(ii),

   $$FV(rhs(L[\tau/\alpha])) \cap FV(S\Gamma) = FV(rhs(L)) \cap FV(S\Gamma) \subseteq FV(S\Delta).$$

   (ii) By induction, $FV(rhs(L), \tau) \cap FV(S\Gamma, f : S\alpha) \subseteq FV(S\Delta)$, and since $rhs(L) = rhs(L[\tau/\alpha])$, this extends to

   $$FV(rhs(L[\tau/\alpha]), \tau) \cap FV(S\Gamma) \subseteq FV(S\Delta).$$

   Since $\tilde{U}$ does not affect $FV(S\Gamma) - FV(S\Delta)$, by Proposition 13 we get

   $$FV(rhs(\tilde{U}(L[\tau/\alpha])), \tilde{U}\tau) \cap FV(\tilde{U}S\Gamma) \subseteq FV(\tilde{U}S\Delta).$$

   (iii) By claim 1., there are residual substitutions $(T_1, \ldots, T_n)$ such that $(Id, T_1, \ldots, T_n)$ solves $\tilde{U}(L[\tau/\alpha])$. So for each i and each $\delta \in FV(\tilde{U}S\Delta)$, $\delta \sqsubseteq_i \delta \in \tilde{U}(L[\tau/\alpha])$ is solved by (Id, T_i), i.e. $T_i\delta = \delta$ is not a pattern variable with respect to T_i. This showed

   $$PV(\tilde{U}(L[\tau/\alpha])) \cap FV(\tilde{U}S\Delta) = \emptyset. \tag{23}$$

   (iv) Suppose $\sqsubseteq_i$ occurs in $\tilde{U}(L[\tau/\alpha])$ and $\delta \in FV(\tilde{U}S\Delta)$. Then $\sqsubseteq_i$ occurs in L, and there is $\delta' \in FV(S\Delta)$ such that $\delta \in FV(\tilde{U}\delta')$. By induction, there is $\tau' \sqsubseteq_i \tau' \in L$ with $\delta' \in FV(\tau')$. Note that $\tau' \sqsubseteq_i \tau' \in L[\tau/\alpha]$, since $\alpha \notin rhs(L)$ as shown under 0.(i). Hence $\tilde{U}\tau' \sqsubseteq_i \tilde{U}\tau' \in \tilde{U}(L[\tau/\alpha])$ and $\delta \in FV(\tilde{U}\tau')$.

   (v) Suppose $y : \overline{\sigma} \in \Gamma$. By induction hypothesis 0.(vi) for e,

   $$FV(L - L^f) \cap FV(S\alpha) \subseteq FV(S\Delta),$$

and since by hypothesis 0.(i) for e, S does not affect $FV(\Gamma, f:\alpha) - FV(\Delta)$, this shows that $\alpha = S\alpha$ does not occur in $L - L^f$. Hence, since $y \neq f$, we have $\alpha \notin FV(L^y)$ and therefore $\tilde{U}(L[\tau/\alpha])^y = \tilde{U}(L^y[\tau/\alpha]) = \tilde{U}(L^y)$.

To show

$$PV(\tilde{U}(L[\tau/\alpha])^y) \subseteq FV(\tilde{U}S\overline{\sigma}),$$

we look at $PV(L^y)$. By the induction hypothesis and 0.(iii) for e, we have

$$PV(L^y) \subseteq FV(S\overline{\sigma}) - FV(S\Delta),$$

and since $\tilde{U}$ does not affect $FV(S\Gamma) - FV(S\Delta)$, it does not affect $PV(L^y)$. By Lemma 12 and Lemma 14, we get

$$\begin{aligned} PV(\tilde{U}(L[\tau/\alpha])^y) &= PV(\tilde{U}(L^y)) = \tilde{U}(PV(L^y)) \\ &\subseteq FV(\tilde{U}S\overline{\sigma}) - FV(\tilde{U}S\Delta) \subseteq FV(\tilde{U}S\overline{\sigma}). \end{aligned}$$

(vi) Suppose $y:\overline{\sigma} \in \Gamma$. Since

$$\tilde{U}(L[\tau/\alpha]) - \tilde{U}(L[\tau/\alpha])^y = \tilde{U}((L - L^y)[\tau/\alpha])),$$

it is sufficient to show

$$FV(\tilde{U}((L - L^y)[\tau/\alpha])) \cap FV(\tilde{U}S\overline{\sigma}) \subseteq FV(\tilde{U}S\Delta). \tag{24}$$

By the induction hypothesis and 0.(ii) for e,

$$FV(L - L^y, \tau) \cap FV(S\overline{\sigma}) \subseteq FV(S\Delta),$$

and since $\tilde{U}$ does not affect $FV(S\Gamma) - FV(S\Delta)$, this gives (24).

1. Since $\tilde{U}$ is a semiunifier of $L[\tau/\alpha]$, $\tilde{U}(L[\tau/\alpha])$ is solved, i.e. has Id as a semiunifier.
2. By the induction hypothesis, for all (S', τ') the following are equivalent:

   (i) $S'\Delta, \overline{S'\Gamma}^{S'\Delta}, f:\overline{S'\alpha}^{S'\Delta} \vdash e:\tau'$

   (ii) for some semiunifier U' of L, $S' =_{\Delta,\Gamma,f:\alpha} U'S$ and $\tau' = U'\tau$.

   (i) $\Rightarrow$ (ii): Let $(S', \{f:\overline{\sigma}\})$ be a typing of $\texttt{val rec}\ f = e$ modulo Δ, Γ. By the typing rule for polymorphic recursion, there is a monotype τ' with

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta}, f:\overline{\sigma} \vdash e:\tau' \quad \text{and} \quad \overline{\sigma} = \overline{\tau'}^{S'\Delta}.$$

   Since α did not occur in Δ, Γ and we can assume that it does not occur in the range of S', we may modify S' so that $S'\alpha = \tau'$. Then (S', τ') is a typing of e modulo $\Delta, \Gamma, f:\alpha$. By induction, there is a semiunifier U' of L such that

$$S' =_{\Delta,\Gamma,f:\alpha} U'S \quad \text{and} \quad \tau' = U'\tau.$$

   Since $S\alpha = \alpha$ by hypothesis 0.(i), one has $U'\alpha = U'S\alpha = S'\alpha = \tau' = U'\tau$, and hence U' is a semiunifier of $L[\tau/\alpha]$. Its most general semiunifier $\tilde{U}$ is a factor of U', i.e. $U' =_{L[\tau/\alpha]} U\tilde{U}$ for some U.

If $f \in FV(e)$, then α occurs in L, so $U'\tau = U\tilde{U}\tau$ and L contains some $\sqsubseteq_i$ with $i \in I(f)$. Since $L \vdash S\Delta \sqsubseteq_i S\Delta$ by induction hypothesis 0.(iv), each $\delta \in FV(S\Delta)$ occurs in an inequation of $L[\tau/\alpha]$. This gives $U' =_{S\Delta} U\tilde{U}$ and hence

$$\overline{U'\tau}^{U'S\Delta} = \overline{U\tilde{U}\tau}^{U\tilde{U}S\Delta}.$$

By our choice of *mgsu* we can assume that $\tilde{U}$ does not affect $FV(S\Gamma) - FV(L[\tau/\alpha])$, and hence that $U' =_{S\Gamma} U\tilde{U}$ as well. Because $U\tilde{U}$ is a semiunifier of $L[\tau/\alpha]$, we have found a semiunifier U of $\tilde{U}(L[\tau/\alpha])$ such that

$$S' =_{\Delta,\Gamma} U\tilde{U}S \quad \text{and} \quad \overline{\sigma} = \overline{\tau'}^{S'\Delta} = \overline{U'\tau}^{U'S\Delta} = \overline{U\tilde{U}\tau}^{U\tilde{U}S\Delta}. \tag{25}$$

If $f \notin FV(e)$, then α does not occur in L, so $L = L[\tau/\alpha]$ and hence $\tilde{U} = Id$ by induction hypothesis 1. Then U' is a semiunifier U of $\tilde{U}(L[\tau/\alpha]) = L$ satisfying (25). Hence (ii) holds.

(ii) $\Rightarrow$ (i): Let U be a semiunifier of $\tilde{U}L[\tilde{U}\tau/\alpha]$ such that $\overline{\sigma} = \overline{U\tilde{U}\tau}^{U\tilde{U}S\Delta}$ and $S' =_{\Delta,\Gamma} U\tilde{U}S$. Then $U' := U\tilde{U}$ is a semiunifier of $L[\tau/\alpha]$. From induction hypotheses 0.(i) and 0.(ii) it follows that $\alpha \notin FV(\tau) \cup FV(S\Gamma, S\Delta)$, and so $\overline{\sigma}$ and S' restricted to $FV(\Delta,\Gamma)$ remain unchanged when modifying U' and S' on α. Hence, redefining $U'\alpha := U'\tau$ and $S'\alpha := U'S\alpha$, U' is a semiunifier of L with $S' =_{\Delta,\Gamma,f:\alpha} U'S$. By the induction hypothesis,

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta}, f : \overline{S'\alpha}^{S'\Delta} \vdash e : U'\tau.$$

Since, by induction hypothesis 0.(i), $S\alpha = \alpha$ and therefore $S'\alpha = U'S\alpha = U'\alpha = U'\tau$, we get $\overline{S'\alpha}^{S'\Delta} = \overline{U'\tau}^{S'\Delta} = \overline{U'\tau}^{U'S\Delta} = \overline{\sigma}$, and so

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta}, f : \overline{\sigma} \vdash e : U'\tau \quad \text{and} \quad \overline{\sigma} = \overline{U'\tau}^{S'\Delta}.$$

By the typing rule for recursive declarations, it follows that $(S', \{f : \overline{\sigma}\})$ is a typing of $\texttt{val rec}\, f = e$ modulo Δ, Γ.

(c) Suppose $\texttt{val rec}\, x = e$ is typable modulo Δ, Γ. Then there are S' and τ' with

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta} \vdash \texttt{val rec}\, x = e : \{x : \overline{\tau'}\}.$$

By the (polymorphic) typing rule for (rec), $\overline{\tau'} = \overline{\tau'}^{S'\Delta}$ and

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta}, x : \overline{\tau'}^{S'\Delta} \vdash e : \tau'.$$

Hence for fresh α we can assume

$$S'\Delta, \overline{S'\Gamma}^{S'\Delta}, x : \overline{S'\alpha}^{S'\Delta} \vdash e : \tau'.$$

By induction, there are L, S and τ such that

$$\mathcal{W}^{+}_{exp}(\Delta, \Gamma, x : \alpha,\ e) = (L, S, \tau).$$

By 2.(ii), there is a semiunifier U of L with

$$S' =_{\Delta,\Gamma} US \quad \text{and} \quad \tau' = U\tau,$$

which is a semiunifier of $L[\tau/\alpha]$, as was shown in proving 2.(ii) from 2.(i). By our assumption about *mgsu*, we have $mgsu(L[\tau/\alpha]) = \tilde{U} \neq fail$, and hence $\mathcal{W}^+_{dec}(\Delta, \Gamma,\ \texttt{val rec}\, x\, \texttt{=}\, e) = (\tilde{U}(L[\tau/\alpha]), \tilde{U}S, \forall\boldsymbol{\beta}\,\tau)$ for suitable $\boldsymbol{\beta}$.

5 Appendix

5.1 Curry-style type inference for polymorphic recursion

Our algorithm $\mathcal{W}^+$ is intended to be implementable as a modification of existing compilers for *SML*, which are based on Milner's $\mathcal{W}$. For comparison, we sketch the ML^+-type inference methods that follow from Henglein[5] and Kfoury[12] characterizations of ML^+-typability in Curry's style. The paper most directly concerned with ML^+ is Henglein's. As it contains a substantial error, we first sketch a correction.

Henglein translates the Milner-Mycroft calculus into a syntax-directed calculus, which is the one given in Figure 1 above except that he uses rec-expressions rather than declarations, with the typing rule (equivalent to)

$$\text{(PolyRec)} \qquad \frac{\Gamma \cup \{x : \overline{\tau}^{\Gamma}\} \vdash e : \tau, \quad \overline{\tau}^{\Gamma} \succ \sigma}{\Gamma \vdash (\texttt{rec}\, x\, \texttt{=}\, e) : \sigma}$$

To obtain a quantifier-free description of Milner-Mycroft-typability, an environment Γ is represented by $(\Gamma', \boldsymbol{\alpha})$, where Γ' is Γ with all type quantifiers removed, and $\boldsymbol{\alpha}$ a sequence of type variables such that $FV(\Gamma) = FV(\Gamma') \cap \boldsymbol{\alpha}$. For each expression e, he defines a *system of equations and inequations*, $SEI(\Gamma', \boldsymbol{\alpha}, e)$ as in Figure 4.

It is then claimed that (Corollary 8 of [5]) when $(\Gamma', \boldsymbol{\alpha})$ represents Γ,

$$\Gamma \vdash_{ML^+} e : \tau \text{ for some } \Gamma, \tau \iff SEI(\Gamma', \boldsymbol{\alpha}, e) \text{ has a semiunifier.} \tag{26}$$

As remarked eralier, this is wrong, since the clause for typing a variable does not take the context of its binding position into account – which is needed for occurrences of recursion variables. [11] This can be seen from the fact that this context occurs in the constraints motivated from the typing rule (See section 2, (2)).

A correction is most easily obtained by revising the notion of environment, so that both (a) scope relations between variable bindings and (b) the difference between monomorphic λ-bindings and polymorphic `rec/let`-bindings can be recovered from an environment.

Let a *scoped (quantifier-free) environment* Γ contain statements $\lambda x : \tau$, for free variables of e that are considered λ-bound, and statements $x : \tau$ for variables x considered `rec-` or `let-bound` in a global term containing e. Let Γ be a *list* of such assumptions, linearly

[11] The statement of Theorem 4 in [5], from which the corollary is derived, also is wrong, even for applicative expressions like $x \cdot 0$, since the substitution is not applied to the context.

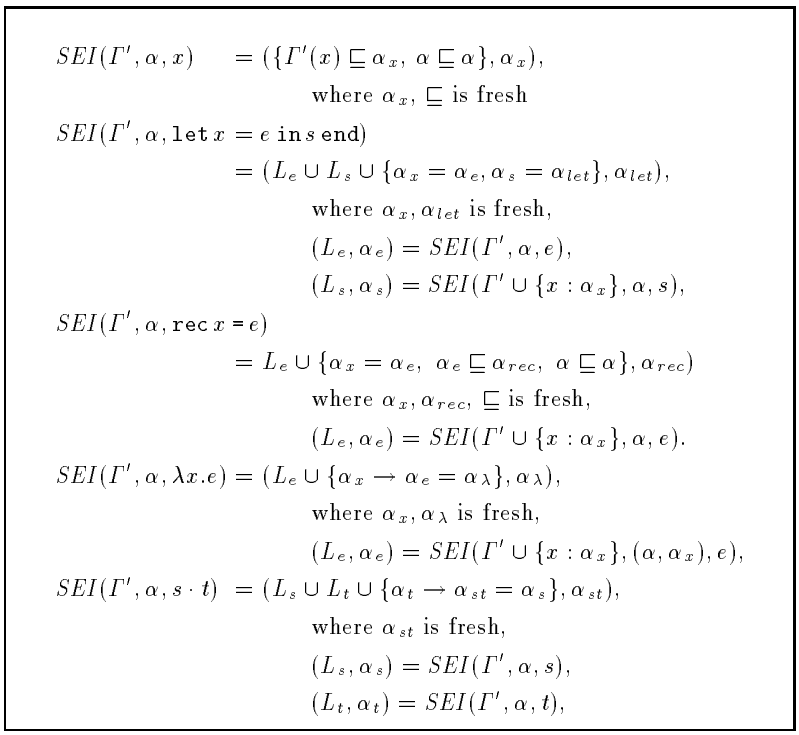

Fig. 4. Henglein's constraints in Curry-style

ordered by $\geq_\Gamma$ from left to right, corresponding to decreasing scope of the respective binding operators in the global term. A new assumption $x : \tau$ (with smallest scope) is appended to Γ at the end, yielding $\Gamma; x : \tau$.

If $\lambda x : \tau \in \Gamma$ is an assumption for a λ-bound variable, then the constraints L, defined via $SEI(\Gamma, x) = (L, \alpha_x)$ when typing an occurrence of x, contain $\tau \sqsubseteq \alpha_x$ and $\tau \sqsubseteq \tau$, which is equivalent to the constraint $\tau = \alpha_x$. Hence λ-bound variables get monomorphic types.

If $x : \tau \in \Gamma$ is an assumption for a `rec-` or `let-`bound variable, the constraints defined for an occurrence of x only say that $\tau \sqsubseteq \alpha_x$ and $\sigma \sqsubseteq \sigma$ for each λ-bound variable $y : \sigma$ having wider scope than x. (For `rec-` or `let-`bound variables y with wider scope than x, we need not demand $\sigma \sqsubseteq \sigma$, since type variables of σ will be suitably quantified in the context $\overline{S\Gamma}^{\lambda S\Gamma}$.)

Let $\overline{S\Gamma}^{\lambda S\Gamma}$ be the ML^+-environment obtained from a scoped environment Γ and a substitution S as the set of the following typing assumptions:

(i) every $\lambda x : \tau$ in Γ is replaced by $\lambda x : S\tau$, and

(ii) every $x : \tau$ in Γ is replaced by $x : \overline{S\tau}^{\lambda S\Gamma, x}$, where $x : \overline{S\tau}^{\lambda S\Gamma, x}$ is obtained from $S\tau$ by

$$
\begin{aligned}
SEI(\Gamma, x) \quad &= (\{\tau \sqsubseteq \alpha_x,\ \alpha \sqsubseteq \alpha\}, \alpha_x), \\
&\qquad \text{where } \alpha_x, \sqsubseteq \text{ is fresh}, \\
&\qquad \alpha = \{\, \sigma \mid \lambda y : \sigma \geq_\Gamma (\lambda) x : \tau \,\} \\
SEI(\Gamma, \lambda x.e) &= (L_e \cup \{\alpha_x \to \alpha_e = \alpha_\lambda\}, \alpha_\lambda), \\
&\qquad \text{where } \alpha_x, \alpha_\lambda \text{ is fresh}, \\
&\qquad (L_e, \alpha_e) = SEI(\Gamma; \lambda x : \alpha_x, e), \\
SEI(\Gamma, s \cdot t) \ &= (L_s \cup L_t \cup \{\alpha_t \to \alpha_{st} = \alpha_s\}, \alpha_{st}), \\
&\qquad \text{where } \alpha_{st} \text{ is fresh}, \\
&\qquad (L_s, \alpha_s) = SEI(\Gamma, s), \\
&\qquad (L_t, \alpha_t) = SEI(\Gamma, t), \\
SEI(\Gamma, \texttt{let}\, x = e\, \texttt{in}\, s\, \texttt{end}) & \\
&= (L_e \cup L_s \cup \{\alpha_x = \alpha_e, \alpha_s = \alpha_{let}\}, \alpha_{let}), \\
&\qquad \text{where } \alpha_x, \alpha_{let} \text{ is fresh}, \\
&\qquad (L_e, \alpha_e) = SEI(\Gamma, e), \\
&\qquad (L_s, \alpha_s) = SEI(\Gamma; x : \alpha_x, s), \\
SEI(\Gamma, \texttt{rec}\, x = e) & \\
&= (L_e \cup \{\alpha_x = \alpha_e,\ \alpha_e \sqsubseteq \alpha_{rec},\ \alpha \sqsubseteq \alpha\}, \alpha_{rec}) \\
&\qquad \text{where } \alpha_x, \alpha_{rec}, \sqsubseteq \text{ is fresh}, \\
&\qquad (L_e, \alpha_e) = SEI(\Gamma; x : \alpha_x, e), \\
&\qquad \alpha = \{\, \sigma \mid \lambda y : \sigma \text{ in } \Gamma. \,\}
\end{aligned}
$$

Fig. 5. Constraints in Curry-style for scoped environments

universally quantifying all its type variables not in

$$\bigcup \{\, FV(S\sigma) \mid \lambda y : \sigma \geq_\Gamma x : \tau \,\}.$$

This modification of Henglein's algorithm is correct in the following sense:

Theorem 16. *Let Γ be a scoped environment, e an expression and $(L, \alpha_e) = SEI(\Gamma, e)$. For any substitution S,*

$$\overline{S\Gamma}^{\lambda S\Gamma} \vdash_{ML^+} e : \tau \quad \Longleftrightarrow \quad S \text{ is a semiunifier of } L \text{ and } \tau = S\alpha_e. \tag{27}$$

Proof (Sketch) We skip the case of variables(!) and only consider the case of a recursive expression. Let $(L, \alpha_{rec}) = SEI(\Gamma, \texttt{rec}\, x = e)$, and $(L_e, \alpha_e) = SEI(\Gamma; x : \alpha_x)$ with fresh α_x.

$\Longleftarrow$: Let S be a semiunifier of L. Since S also is a semiunifier of L_e, by induction we have

$$\overline{S(\Gamma; x : \alpha_x)}^{\lambda S(\Gamma; x:\alpha_x)} \vdash_{ML^+} e : S\alpha_e.$$

Note that

$$\overline{S(\Gamma; x:\alpha_x)}^{\lambda S(\Gamma; x:\alpha_x)} = \overline{S\Gamma}^{\lambda S\Gamma} \cup \{x : \overline{S\alpha_x}^{\lambda S\Gamma, x}\}.$$

Let $\overline{\Gamma} := \overline{S\Gamma}^{\lambda S\Gamma}$ and $\tau := S\alpha_x = S\alpha_e$. Note that $\overline{\tau}^{\overline{\Gamma}} = \overline{\tau}^{\lambda S\Gamma, x}$, and so

$$\overline{\Gamma} \cup \{x : \overline{\tau}^{\overline{\Gamma}}\} \vdash_{ML^+} e : \tau.$$

Since S solves $\alpha_e \sqsubseteq \alpha_{rec}$ and $\sigma \sqsubseteq \sigma$ for all $\lambda y : \sigma \geq_\Gamma x : \alpha_x$ (with the same residual matcher), we also have $\overline{\tau}^{\overline{\Gamma}} \succ S\alpha_{rec}$. An application of the typing rule (PolyRec) gives

$$\overline{\Gamma} \;=\; \overline{S\Gamma}^{\lambda S\Gamma} \vdash_{ML^+} \texttt{rec}\, x = e : S\alpha_{rec}. \tag{28}$$

$\Longrightarrow$: Conversely, if (28) holds, by the typing rule (PolyRec) there is some τ such that

$$\overline{\Gamma} \cup \{x : \overline{\tau}^{\overline{\Gamma}}\} \vdash_{ML^+} e : \tau \quad \text{and} \quad \overline{\tau}^{\overline{\Gamma}} \succ S\alpha_{rec}.$$

We find that

$$\overline{\Gamma} \cup \{x : \overline{\tau}^{\overline{\Gamma}}\} = \overline{S(\Gamma; x:\alpha_x)}^{\lambda S(\Gamma; x:\alpha_x)},$$

so by induction, S is a semiunifier of L_e and $\tau = S\alpha_e$. Because $\overline{\tau}^{\overline{\Gamma}} \succ S\alpha_{rec}$, there is a substitution T instantiating bound quantifiers of $\overline{\tau}^{\overline{\Gamma}} = \overline{S\alpha_e}^{\overline{\Gamma}}$ so that $T\tau = TS\alpha_e = S\alpha_{rec}$. Since the variables free in $\overline{\Gamma} = \overline{S\Gamma}^{\lambda S\Gamma}$ are mapped to themselves, we have $TS\sigma = S\sigma$ for each $\lambda y : \sigma$ in Γ. Hence (S, T) solve the inequations $\alpha_e \sqsubseteq \alpha_{rec}$ and $\sigma \sqsubseteq \sigma$ for $\lambda y : \sigma$ in Γ. From $x : \overline{\tau}^{\overline{\Gamma}} = x : \overline{S\alpha_e}^{\overline{\Gamma}} = x : \overline{S\alpha_x}^{\lambda S(\Gamma; x:\alpha_x), x}$ we also get $S\alpha_x = S\alpha_e$. Thus, S is a semiunifier of L. □

From the equivalence (27) a principal types property follows: If $(L, \alpha_e) = SEI(\Gamma, e)$ and U is the most general semiunifier of L, then $U\alpha_e$ must be a principal ML^+-type of e 'modulo the scoped environment Γ' (in a suitable sense).

To typecheck a term using an equivalence like (27), there are two ways to proceed:

(i) Process the entire term first, compute its constraint system, find a most general semiuinfier (if possible), and compute from it the principal type and environment modification. This method has the drawback that type errors can hardly be localized from the full constraint set of the term.

(ii) Process the term in a fixed order and for each subterm you meet, compute its constraint system, try to find its semiunifier, compute its principal type and corresponding modification of the environment, and with modified environment, go on to the next subterm. This method allows one to report type errors for untypable subterms, relative to their environment determined by the traversal – just as it is done in ML.

In the ML case, however, the solution of two subproblems can efficiently be combined, while in the case of ML^+ this is not obviously possible, due to properties of semiunification: Suppose $(L_1, \alpha_1) = SEI(\Gamma, e_1)$ with $S_1 = mgsu(L_1)$, for a first subexpression, followed by $(L_2, \alpha_2) = SEI(S_1\Gamma, e_2)$ with $S_2 = mgsu(L_2)$. Then S_1L_1 is solved, but applying the solution S_2 of L_2 might make S_1L_1 unsolved: S_2S_1 need not be a solution of L_1 (cf. remark 1 on p.8. If L_1 is equational then S_2S_1 is a solution of L_1). As a remedy, one might explicitly call *mgsu* to solve $S_2S_1L_1 \cup S_2L_2$. This would have the drawback of calling the potenially expensive *mgsu* at every node of a term.

These methods of type inference for polymorphic recursion follow from both Henglein[5] and Kfoury e.a[12]. The most important advantage of our $\mathcal{W}^+$ is the following: the complex proof of Theorem 10 shows that one can have the benefit of localized error reports (from typing subterms in turn as in (ii)), but that –for a particular traversal of terms– no additional calls to *mgsu* as in (ii) are needed: inequational constraints have to be solved only at recursive declaration-subterms. Perhaps these advantages could also be obtained with an adaption of the Curry-style approach; but then the correctness proof would probably involve the subtleties seen in Theorem 10.

Other differences are that we treat monomorphic and polymorphic variables (in contrast to [12]), and expressions and declarations (in contrast to [5] and [12]).

5.2 A variation of $\mathcal{W}^+_{exp}$ exploiting scope information

Our algorithm $\mathcal{W}^+$ uses *sets* Δ, Γ of typing assumptions to represent the environment $\Delta \cup \overline{\Gamma}^{\Delta}$. This has the drawback that when typing a variable occurrence of a polymorphic assumption $y : \overline{\sigma} \in \Gamma$, the monomorphic part Δ of the enviroment does not distinguish between assumptions $x : \tau$ where x has wider scope than y from assumptions $z : \rho$ where z has smaller scope than y - a distinction used in the constraints for the typing rule for polymorphic recursions. Consequently, in the var-clause of $\mathcal{W}^+_{exp}$ one may add constraints $\delta_z \sqsubseteq_i \delta_z$ that are unnecessary, and in the λ-clause therefore has to subtract some.

Therefore, it seems that considering the environment as a *set*, though appropriate for an analysis of *ML*-typing and –as shown above– still possible for ML^+, is not the proper approach in a Milner-style type inference procedure for ML^+. Had we scope information available (for example, by considering the environment as a list), then one might add less constraints in the var-clause and avoid the substraction of constraints in the λ-clause. The corresponding clauses of $\mathcal{W}^+$ would have to be replaced by:

Case $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ x) =$

$$\begin{cases} (\emptyset, Id, \tau), & \text{if } \Delta(x) = \tau, \\ (L, Id, \tau[\boldsymbol{\alpha}'/\boldsymbol{\alpha}, \boldsymbol{\beta}'/\boldsymbol{\beta}]), & \text{if } \Gamma(x) = \forall \boldsymbol{\beta}\tau, \text{ where} \end{cases}$$

$\boldsymbol{\alpha} = FV(\forall \boldsymbol{\beta}\tau) - FV(\Delta)$,
$\boldsymbol{\alpha}' =$ new copies of $\boldsymbol{\alpha}$,
$\boldsymbol{\beta}' =$ new copies of $\boldsymbol{\beta}$,
$L = \{\alpha \sqsubseteq_i \alpha' \mid \alpha \in \boldsymbol{\alpha}\} \cup \{\delta \sqsubseteq_i \delta \mid \delta \in FV(\Delta_x)\}$,
with a new relation $\sqsubseteq_i$, and i added to $I(x)$, where
$\Delta_x = \{\, z : \sigma \in \Delta \mid z \text{ has wider wider scope than } x \,\}$

Case $\mathcal{W}^+_{exp}(\Delta, \Gamma,\ (\texttt{fn}\, x \texttt{=>}\, e)) = (L, S, S\alpha \rightarrow \tau)$,
if $(L, S, \tau) = \mathcal{W}^+_{exp}(\Delta, x : \alpha, \Gamma,\ e)$, where α is a fresh variable.

Some changes in the claims of Theorem 10 seem to be necessary, in particular:

1. Change $\Delta, \overline{\Gamma}^{\Delta}$ by putting $\overline{\Gamma}^{\Delta} := \{\, x : \overline{\sigma}^{\Delta_x} \mid x : \sigma \in \Gamma \,\}$.

2. Instantiate $\Gamma(x) = \forall \boldsymbol{\beta} \tau$ by copying only $\boldsymbol{\alpha} = FV(\forall \boldsymbol{\beta} \tau) - FV(\Delta_x)$.

Additional changes in the auxiliary claims 0.i) - 0.vi) of the main proof will be necessary as well. Though we have not considered a modification of Theorem 10 in detail, the case of λ-abstractions can be expected to be much simpler.

While this might lead to simplifications of the proof, one also needs an efficient implementation environments with scope information. Our representation of an environment by partial functions reflects what is done in the compiler of SML of New Jersey, where type information is stored in updatable reference cells of variables. No relational information (like scope inclusions) is directly available.

5.3 Implementation details

The compiler of *SML of New Jersey*[1] proceeds in several phases. In the first phase, an abstract syntax tree of the term to be typed is constructed by the parser, which introduces fresh names for individual variables. Hence we can assume that free and bound variables are disjoint and no variable is bound twice.

The next phase consists of an elaboration of the abstract syntax tree; essentially, a fresh type variable α_x is stored in a cell referenced by all occurrences of x.

The third phase is the computation of a principal type for the given expression or declaration. While traversing the term, principal types of subterms are computed using $\mathcal{W}$, as well as substitutions of types for type variables. These substitutions are realized as updates of the type reference cells. Note that there is no explicit *set* of typing assumptions maintained. Instead, the environment exists implicitly only, as the position of a subterm and the values of the type reference cells.

Quantification of type variables is controlled by their depth, an updatable attribute of type variables. The depth of α_x is set to the nesting depth of the binding position of x. When a type variable α is unified with a type expression σ, the depth of α and each type variable β of σ is adjusted to $\min\{depth(\alpha), depth(\beta)\}$.

In the case of a declaration $d = \texttt{val}\ x\ \texttt{=}\ e$ or $d = \texttt{val rec}\ x\ \texttt{=}\ e$, quantification of type variables is realized as follows. If τ is the computed principal type of e and $S\Gamma$ its environment, we need $x : \overline{\tau}^{S\Gamma}$ as additional assumption when typing the scope of d. The context $S\Gamma$ is available as the position of d in the global term and the contents of the reference cells of variables bound on the path to d. To find which of the free type variables of τ have to be quantified to give $\overline{\tau}^{S\Gamma}$, once simply compares their depth with the depth of the the binding position d of x; those with larger depth have to be quantified.

The type-checker of *SML of New Jersey* has two disjoint kinds of type variables, free and bound ones. Therefore, type quantifiers need not be written explicitly. When a type variable has to be quantified, it is replaced be a fresh 'bound' one. Instantiation of 'bound' type variables in typing an occurrence of x is done by replacing the bound variables of the assumed type $x : \forall \boldsymbol{\beta}\, \tau$ by fresh 'free' type variables $\boldsymbol{\beta}'$ in τ (with depth infinity.)

To implement $\mathcal{W}^+$, the following modifications have been made:

Variables To type an occurrence of a variable x with assumption $x : \overline{\tau}$ in context Δ, Γ, we need $FV(\overline{\tau}) - FV(\Delta)$, i.e. we must know which type variables occur in an assumption type of a monomorphic individual variable. We simply collect the assumption types of λ-bound individual variables in a list, the λ- or *rule-bound types*. On lookup in this list, the types are decomposed and replaced by the type variables they contain. Since lookup in a list is slow, a better solution would be to add an attribute to type variables; we hesitated to do so in order not to modify the *SML*-type of *type* resp. *type variable*.

Declarations For declarations `val` x `=` e with derived type τ for e in the context $S\Delta, S\Gamma$, we need to return a suitably quantified typing $x : \tau$. While for *ML* we have $\overline{\tau} = \overline{\tau}^{S\Delta}$, for ML^+ the special type variables (those in $spec(S\Gamma, L)$) must not be quantified. These are identified via their depth attributes. (On creating fresh copies of type variables when typing a variable occurrence, the copy gets the same depth as its original.) Similarly for recursive declarations, with $U\tau$ and $spec(US\Gamma, UL[U\tau/\alpha])$ instead.

Inequations We use a very simple representation of an inequation constraint set: a list of triples (σ, i, τ) for $\sigma \sqsubseteq_i \tau$. To find a most general semiunifier, reduction rules (like those of Henglein[5]) are applied in a certain order, until the system is in a solved form that allows one to read off the most general semiunifier. The treatment of inequations $\tau \sqsubseteq_i \alpha$ involves a 'generalized occurs check' to exclude chains $\alpha \sqsubseteq_j \ldots \sqsubseteq_k \tau \sqsubseteq_i \alpha$ in L with some non-variable type; this is a global, hence potentially expensive test.

Although semiunification is undecidable, examples where this procedure loops remain to be constructed. (The number of variables in a system may grow under reductions.)

Abstraction When typing an abstraction (`fn` x `=>` e) according to $\mathcal{W}^+$, we have to adjust the constraint set L delivered in $\mathcal{W}^+(\Delta \cup \{x : \alpha\}, \Gamma, e) = (L_\alpha, S, \tau)$. First we have to identify the inequations of the form $\tau \sqsubseteq_i \tau$ in L_α, where i represents an occurrence of a polymorphic individual variable in the context, i.e. $i \in I(y)$ with some $y : \overline{\tau} \in \Gamma$. Then $\tau \sqsubseteq_i \tau$ has to be replaced by all $\gamma \sqsubseteq_i \gamma$ with $\gamma \in FV(\tau), \gamma \notin FV(S\alpha) - FV(S\Delta)$.

The current implementation described in Emms[3] does not perform the first step in the case of abstractions properly, as was revealed when carrying out this proof. [12]

A proper treatment would give an additional attribute to individual variables, the nesting depth d of their binding. This would be passed into inequations when typing variable occurrences, giving $\tau \sqsubseteq_{i,y(d)} \tau$. If (`fn` x `=>` e) has binding depth d_x and $d < d_x$, $\tau \sqsubseteq_{i,y(d)} \tau$ in L_α would be replaced by $\gamma \sqsubseteq_{i,y(d)} \gamma$ for the type variables $\gamma \notin FV(S\alpha) - FV(S\Delta)$.

5.4 An example of inferring a type with $\mathcal{W}^+$

We demonstrate our implementation on an example, due to Stefan Kahrs and communicated by Thorsten Altenkirch. The idea is to define the set n-`Lam` of λ-terms with free variables among $v_1, \ldots, v_n$. Since SML does not have types depending on individual terms, the number n is coded as an n-fold iteration of a type constructor.

[12] It tries to identify the relevant variables through the attributes available in datatypes as they are in the compiler. It is correct at least for terms where recursive declarations occur at top level and do not contain further declarations.

If type `'a` represents an initial segment $\{1,\ldots,n\}$ of numbers, type `'a Lift` represents $\{1,\ldots,n+1\}$, consisting of a `new` element $n+1$ and `old` elements $1,\ldots,n$:

```
datatype 'a Lift =
    new | old of 'a;
```

To define a datatype `'a Lam` of λ-terms in $v_1,\ldots,v_n$, we need an increasing type parameter: if t's free variables are among $\{v_1,\ldots,v_{n+1}\}$, those of $\lambda v_{n+1}.t$ are among $\{v_1,\ldots,v_n\}$:

```
datatype 'a Lam =
    var of 'a
  | app of ('a Lam) * ('a Lam)
  | abs of ('a Lift) Lam;
```

To define substitutions, suppose for indices k of type `'a` we have λ-terms $f(k)$ whose free variables have indices j of type `'b`. The substitution of the v_k by $f(k)$ in an `'a Lam`-term t would then be defined by `bindLam f t`:

```
fun bindLam f (var x) = f x
  | bindLam f (app (t,u)) = app (bindLam f t,bindLam f u)
  | bindLam f (abs t) = abs (bindLam (liftLam f) t)
and liftLam f new = var new
  | liftLam f (old x) = bindLam (var o old) (f x);
```

Since `bindLam` is applied to the arguments `f` and `liftLam f`, which are of different type, this recursive definition is untypable in *SML*. The following trace, extended by some comments, shows how the types

```
bindLam = fn : ('a -> 'b Lam) -> 'a Lam -> 'b Lam
liftLam = fn : ('a -> 'b Lam) -> 'a Lift -> 'b Lift Lam
```

are inferred by our extended type checker for SML^+. To simplify the presentation, in we omit the case of applications.

For the constructors, SML^+ infers a context `Gamma`.

Phase 1: From unknown polytype assumptions, infer approximate type kernels and inequation constraints

```
W+(Delta,Gamma+{ bindLam :'a_bind, liftLam :'a_lift },
   (body_bind,body_lift)) = (S, L, (tau_bind,tau_lift)),
```
where

```
S = Id on assumed types,

L = ('a_bind  <18  ('X -> 'X Lift Lam)
                              -> 'W -> 'V Lift Lam)
    ('a_lift  <7  ('Z -> 'Y Lam) -> 'T)
    ('a_bind  <6  'T -> 'Z Lift Lam -> 'Y Lift Lam)

tau_bind = ('Z -> 'Y Lam) -> 'Z Lam -> 'Y Lam
tau_lift = ('U -> 'W) -> 'U Lift -> 'V Lift Lam
```

Phase 2: Solve `L[tau_bind/'a_bind, tau_lift/'a_lift]`

```
(('Z -> 'Y Lam) -> 'Z Lam -> 'Y Lam
       <18  ('X -> 'X Lift Lam) -> 'W -> 'V Lift Lam)

(('U -> 'W) -> 'U Lift -> 'V Lift Lam
               <7  ('Z -> 'Y Lam) -> 'T)

(('Z -> 'Y Lam) -> 'Z Lam -> 'Y Lam
               <6  'T -> 'Z Lift Lam -> 'Y Lift Lam)
```

By decomposition to subterms:

```
('Z  <18  'X)
('Y  <18  'X Lift)
('Z Lam  <18  'W)
('Y  <18  'V Lift)

('U  <7  'Z)
('W  <7  'Y Lam)
('U Lift -> 'V Lift Lam  <7  'T)

('Z -> 'Y Lam  <6  'T)
('Z  <6  'Z Lift)
('Y  <6  'Y Lift)
```

By unifying rh-sides of equal lh-sides in `<18` (`'X ='V`):

```
('Z  <18  'V)
('Y  <18  'V Lift)
('Z Lam  <18  'W)

('U  <7  'Z)
('W  <7  'Y Lam)
('U Lift -> 'V Lift Lam  <7  'T)
```

```
('Z -> 'Y Lam  <6  'T)
('Z  <6  'Z Lift)
('Y  <6  'Y Lift)
```

By expanding `'W` to `'S Lam` in `<18` and `<7` and decomposing:

```
('Z  <18  'V)
('Y  <18  'V Lift)
('Z  <18  'S)

('U  <7  'Z)
('S  <7  'Y)
('U Lift -> 'V Lift Lam  <7  'T)

('Z -> 'Y Lam  <6  'T)
('Z  <6  'Z Lift)
('Y  <6  'Y Lift)
```

By unifying rh-sides of equal lh-sides in `<18`: (`'V='S`)

```
('Z  <18  'S)
('Y  <18  'S Lift)

('U  <7  'Z)
('S  <7  'Y)
('U Lift -> 'S Lift Lam  <7  'T)

('Z -> 'Y Lam  <6  'T)
('Z  <6  'Z Lift)
('Y  <6  'Y Lift)
```

By expanding `'T` to (`'Q Lift -> 'R Lift Lam`) in `<7, <6` and decomposing:

```
('Z  <18  'S)
('Y  <18  'S Lift)

('U  <7  'Z)
('S  <7  'Y)
('S  <7  'R)
('U  <7  'Q)

('Z  <6  'Q Lift)
('Y  <6  'R Lift)
('Z  <6  'Z Lift)
('Y  <6  'Y Lift)
```

By unifying rh-sides of equal lh-sides in `<7 ('Z = 'Q, 'Y ='R)` and removinq duplicates in `<6` we get the solved system :

```
('Q  <18  'S)
('R  <18  'S Lift)

('U  <7  'Q)
('S  <7  'R)

('Q  <6  'Q Lift)
('R  <6  'R Lift)
```

The semiunifier of the initial system is the composition of substitutions made, i.e.

```
sU = ('Z := 'Q, 'Y :='R)
     o ('T := 'Q Lift -> 'R Lift Lam)
     o ('V := 'S) o ('W := 'S Lam) o ('X := 'V)
```

The inferred types of `bindLam,liftLam` are obtained by applying `sU` to

```
tau_bind = ('Z -> 'Y Lam) -> 'Z Lam -> 'Y Lam
tau_lift = ('U -> 'W) -> 'U Lift -> 'V Lift Lam
```

and then generalizing relative to `sUDelta,sUGamma`. Hence:

```
bindLam = fn : ('a -> 'b Lam) -> 'a Lam -> 'b Lam
liftLam = fn : ('a -> 'b Lam) -> 'a Lift -> 'b Lift Lam
```

5.5 Trace of detecting an untypability with $\mathcal{W}^+$

We give an example of a nested recursion that is untyptble in ML^+. The user may choose between inspecting a full trace of the inequation solving and an abbreviated form. In the abbreviated form, for each recursive declaration, the inequation system before and after solutionis presented. The list of type variables in the inequations that occur in assumptions of λ-bound individual variables is shown as the 'rulebound' list - following the terminology of *SML* where 'rules' of the form '`pattern of argument => result`' can be used instead of simple variable binding by λ.

Consider the following nested recursion:

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

Typing the inner recursion, assuming {`f:'Z, x:'W`}, gives:

```
val rec g = (fn y => <exp> <exp>)

initial problem
('Z  <2  'Y -> 'X)      (* from (f:'Z  y:'Y) :'X *)
the rulebound: 'W

solved system is
('Z  <2  'Y -> 'X)
the rulebound: 'W
g:'Y -> 'X              (* dec : { g:{'Y->'X} }  *)
```

Although `'Y,'X` do not occur in the environment, they are not quantified, since they are *specializations of* the unknown poly-assumption `f:'Z`. However, the types `'Y,'X` are in the following treated as polytypes, not as monotypes.

Now type the outer recursion:

```
val rec f = (fn x => let <dec>  in <exp> end)

initial problem
('X  <7  'V)            (* (g:'Y->'X  x:'U):'V  *)
('Y  <7  'U)
('X  <6  'T)            (* (g:'Y->'X  1:int):'T *)
('Y  <6  int)
('U -> 'T * 'V  <2  'Y -> 'X)
                        (* (fn x => let ... end): 'U -> 'T * 'V  *)
the rulebound:

Error: extended occurs check fail in semiunify
  problem inequation
  ('Z * 'Y  <2  'X)

  in current system
  ('X  <7  'Y)
  ('W  <7  'V)
  ('X  <6  'Z)
  ('W  <6  int)
  ('V -> 'Z * 'Y  <2  'W -> 'X)

  in declaration:
    val rec f = (fn x => let <dec>  in <exp> end)
```

The problematioc inequation is obtained by decomposing the last one in the system. Combined with the first, the type checker detects that via `'Z * 'Y <2 'X` and `'X <7 'Y`, the variable `'Y` occurs as a proper 'subterm' of itself; hence the system cannot have a solution by type expressions.

6 Conclusion

We have presented an extension of Damas-Milner type inference to Milner-Mycroft type inference that is fairly close to the code of type checkers in existing compilers for *SML*.

The correctness and weak completeness proof is relatively subtle and more complicated than those for methods in Curry's style. The main reason is that we separate inequational and equational constraints in such a way that equational constraints can be executed as updates of type reference cells during type inference, while inequational constraints are *only* solved when typing a recursively declared variable, and automatically remain solved in all other cases – in particular on applying updates. This allows type inference with polymorphic recursion in an incremental way, with inequation solving localized as much as possible.

From a logical point of view, an inductive proof of the principal types property is complicated because when comparing typing derivations, a less general derivation need not just be a parametric instance of a given one, but may additionally have weaker universal assumptions.

Acknowledgement: An implementation of semiunification (by Robert Stärk) and the implementation of polymorphic recursion for SML of New Jersey, Version 0.93, (by Martin Emms) were supported by the Deutsche Forschungsgemeinschaft, Le 788/1-1.

The theoretical work was supported by the European Community under Esprit Basic Research Action 7232 - 'Common Foundation of Functional and Logic Programming Languages'.

Notice: MIRANDA is a trademark of Research Software, Ltd.

References

1. D. MacQueen e.a. Standard ML of New Jersey, Version 0.93. AT& T Bell Laboratories, 1993.
2. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
3. M. Emms. Documentation for polyrec_sml: An extension of SML with typechecking for polymorphic recursion. Technical Report CIS-Bericht-96-88, Universität München, Centrum für Informations- und Sprachverarbeitung, 1996.
4. F. Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Courant Institute of Mathematical Sciences, May 1989. Technical Report 443, Computer Science Department, New York University.
5. F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15:253–289, 1993.
6. J. Herbrand. Recherches sur la theorie de la demonstration. In *Ecrits logiques de Jacques Herbrand*. PUF, Paris, 1968. thèse de Doctorat d'Etat, Université de Paris (1930).
7. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
8. D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. In *Proceedings of the 8th Conference on Foundations of Software Technology and Theoretical Computer Science. Pune, India, December 21 - 23, 1988*, pages 435 – 454. Springer LNCS 338, 1988.

9. A. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Coll. on Trees in Algebra and Programming (CAAP), Copenhagen, Denmark*, pages 206–220. Springer, May 1990. Lecture Notes in Computer Science, Vol. 431.

10. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML-typability. *Journal of the ACM*, 199?

11. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *22nd Annual ACM Symposium on Theory of Computing. Baltimore, Maryland*, pages 468 – 476, May 1990.

12. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

13. H. Leiß. Semi-unification and type inference for polymorphic recursion. Bericht INF2-ASE-5-89, Siemens AG, München, May 1989.

14. H. Leiß. Polymorphic Recursion and Semi-Unification. In E. Börger, H. Kleine-Büning, and M. M. Richter, editors, *CSL '89. 3rd Workshop on Computer Science Logic. Kaiserslautern, FRG, October 2–6, 1989*, pages 211–224. Springer LNCS 440, 1990.

15. H. Leiß and F. Henglein. A decidable case of the semi-unification problem. In A. Tarlecki, editor, *16th International Symposium on Mathematical Foundations of Computer Science. Kazimierz Dolny, Poland, Sept. 1991*, pages 318 – 327. Springer LNCS 520, 1991.

16. H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Prooceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 382–401. ACM, January 1990.

17. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

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

19. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

20. A. Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming. 6th Colloquium. Toulouse, April 17–19, 1984*, pages 217–228. Springer LNCS 167, 1984.

21. P. Pudlák. On a unification problem related to Kreisel's conjecture. *Commentationes Mathematicae Universitatis Carolinae*, 29(3):551–556, 1988.

22. J. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

23. D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings of the IFIP International Conference on Fuctional Programming Languages and Computer Architecture*. Springer LNCS 201, 1985.