ML with PTIME complexity guarantees*

Jacek Chrząszcz and Aleksy Schubert

University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
{chrzaszcz,alx}@mimuw.edu.pl

Abstract

Implicit Computational Complexity is a line of research where the possibility to inference a valid property for a program implies that the program runs in particular complexity class. Soft type systems are one of the research threads within the field. We present here a soft type system with ML-like polymorphism that enjoys decidable typechecking, type inference and typability problems and gives polynomial time computational guarantees for the running time of typed programs.

1998 ACM Subject Classification F.3.3 Studies of program constructs, F.4.1 Mathematical logic

Keywords and phrases implicit computational complexity, polymorphism, soft type assignment

Digital Object Identifier 10.4230/LIPIcs.CSL.2012.198

1 Introduction

The design of a programming language may be focused on guarantees the language gives to a programmer or a software consumer. Implicit Computational Complexity studies machine-free methods to characterise particular complexity class, e.g. PTIME, NP, PSPACE. This line of research may lead not only to an interesting programming language, but also can give new insights to the theoretical analysis of the subject class.

Soft type systems proposed by Gaboardi et al. [11, 12] that emerged from the Soft Linear Logic (SLL) of Lafont [19], are one of the proposals which makes the expected guarantees with relatively low annotation burden. One drawback of the soft type systems is that they use full second-order polymorphism to gain necessary uniformity of representation [20]. This, however, results in undecidable type-checking and type inference [6]. We regain the necessary uniformity by introduction of special constants similar in fashion to the ones used in [8] to obtain the completeness.

Linear logic brought many characterisations of complexity classes in addition to the well established proposals such as [5, 7, 9, 18, 23, 27], to mention few. This was started by Girard in [16] where the Light Linear Logic (LLL) and Elementary Linear Logic (ELL) were proposed to characterise polynomial and elementary time complexities respectively by means of the cut elimination procedure. The ideas of LLL were taken up by Asperti and Roversi [1] who designed a more flexible affine variant of Girard's logic called Light Affine Logic. A type system which is based on these ideas was presented in [4]. Another line of research on linear logic and complexity classes was started by Lafont's Soft Linear Logic (SLL) [19] which characterises polynomial time complexity and is the starting point for the Soft Type Assignment (STA) systems by Gaboardi et al. [11, 12, 14] where certain form of typability guarantees reduction of lambda terms in polynomial time. The light logic principles have been used to characterise other interesting complexity classes for instance [25] uses Light

* This work was partially supported by the Polish government grant no N N206 355836.

Linear Logic with additional operation + to characterise NP; LOGSPACE is characterised by different versions of Stratified Bounded Affine Logic (SBAL) in e.g. [30, 21].

The type systems that are based on linear logic employ linear modalities (e.g. ! or §) to guard the necessary restrictions. The modalities control duplication of data by marking in the type the particular function argument that is multiplied during the computation. This has effect similar to the one obtained by Bellantoni, Cook and Leivant, but in a way which results from more basic assumptions. For instance, their restriction is obtained by Baillot et al. in [2] due to prefixing of arguments with § and ! in Def. 5. Similar function has the prefixing with ! in STA by Gaboardi and Ronchi Della Rocca [14].

In this paper we present a version of STA [14] with ML-like polymorphism and useful data types such as booleans, integers and strings. The ambition of the paper is similar to the one of [2] to present a contribution close to real language. However, we move the focus here to polymorphism which is not present in the contribution of Baillot et al. nor in earlier papers on monomorphic calculi [10, 6]. The ML polymorphism in our setting is presented in a traditional way which contrasts with the presentation of [22] where a division into upper class and lower class types is used. Moreover, we use linear equations over natural numbers to express the necessary constraints, which is conceptually simpler to the approach by Dal Lago, Schöpp where E-unification is used or the approach by Baillot, Martin [3] where additional disequality constraints are used.

We observe that the set of obtained functionals, in addition to the running time guarantees, provides a natural way to program in a design pattern present in imperative programming. When the pattern is followed all the memory is allocated before any essential computation is done. In this way the dynamic allocation is no longer needed in the course of computation. This way of programming is advised both by the Java Card manufacturers, see e.g. [15, Sect. 2.4.3], and by software verification community, see e.g. [26, Sect. 3].

This paper is structured as follows. We present the syntax and semantics of the system in Sect. 2. Then we explain the primitives of the language in Sect. 3. Basic properties of MLSTA are presented in Sect. 4. The complexity guarantees of the system are proved in Sect. 5 while the decidability of type related problems in Sect. 6. We conclude in Sect. 7.

2 ML-like system MLSTA

We propose a type system which makes possible a more uniform treatment of input data. The system is inspired by ML and uses several algebraic types. Its syntax and types are defined as follows:

$$\begin{array}{rll} A ::= \alpha \mid \sigma \multimap A \mid A \otimes B \mid \mathbb{S}_i^{!j} \mid (\mathbb{S}_i^{!j})^k \boxtimes A \mid \mathbb{N}^{!j} \mid \mathbb{B} & \text{(Linear Types)} & (1)\\ \mathfrak{s} ::= !^i \forall \vec{\alpha}.A & \text{(Type Schemes)} & \\ \sigma ::= !^i A & \text{(Types)} & \\ M ::= x \mid \lambda x.M \mid M_1 M_2 \mid \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 \mid c & \text{(Terms)} & \end{array}$$

where $\alpha \in \mathcal{V}$, which is a countable set of type variables, $i, j, k \in \mathbb{N}$, i.e. natural numbers, with $!^0$ meaning no ! at all, and c is a constant from the set $\mathrm{Const}_{\mathrm{MLSTA}}$ listed in Fig. 2. The set of linear types generated from the nonterminal A above is denoted $\mathcal{T}_A$, the set of type schemes generated from the nonterminal $\mathfrak{s}$ is denoted $\mathcal{T}_\mathfrak{s}$, and the set of types generated from the nonterminal σ is denoted $\mathcal{T}_\sigma$. The contexts in this system are sets of pairs $x : \sigma$ or $x : \mathfrak{s}$.

The reduction relation contains β rules and δ rules presented in Fig. 2. The presentation of the typing rules requires a notion of a closure with respect to a context. For a type A

and a context Γ we define the *closure* of A with respect to Γ as $\mathrm{Clos}(\Gamma; A) = \forall\alpha_1, \ldots, \alpha_n.A$ where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(A)\backslash\mathrm{FTV}(\Gamma)$. The typing rules of the system are presented in Fig. 1. To express that $\Gamma \vdash M : A$ is derivable in MLSTA we write $\Gamma \vdash_{\mathrm{MLSTA}} M : A$. We introduce a succinct notation for derivations inspired by the Church-style form:

$$\mathfrak{D} ::= x^A \mid x^{A\leq\mathfrak{s}} \mid \langle\mathfrak{D}, x : A\rangle^w \mid \lambda x^\sigma.\mathfrak{D} \mid \mathfrak{D}_1\mathfrak{D}_2 \mid \langle\xi x_1, \ldots, x_n : \sigma.\mathfrak{D}, x : !\sigma\rangle^m \mid \langle\mathfrak{D}\rangle^{sp} \mid \\ \texttt{let}\ x^{!^i\forall\vec{\alpha}.B} = \mathfrak{D}_1\ \texttt{in}\ \mathfrak{D}_2$$

The subsequent cases in the above definition are in direct correspondence with the rules presented in Fig. 1. This correspondence makes possible to represent uniquely derivations in MLSTA with the terms defined above (still some terms generated with the grammar above have no corresponding derivation in MLSTA). We sometimes write $\langle\mathfrak{D}, \vec{x} : \vec{A}\rangle^{wn}$ to denote n-times application of the rule (w) with pairs $x_1 : A_1, \ldots, x_n : A_n$. In case a derivation $\mathfrak{D}$ ends with a judgement $\Gamma \vdash M : \sigma$ then we let $\mathfrak{D}_{term} = M$, $\mathfrak{D}_{ctxt} = \Gamma$, and $\mathfrak{D}_{type} = \sigma$.

$$\frac{\mathfrak{s} \geq A}{x : \mathfrak{s} \vdash x : A}\ (Ax) \qquad \frac{x : \mathfrak{s} \in \mathrm{Const}_{\mathrm{MLSTA}} \quad \mathfrak{s} \geq A}{\vdash x : A}\ (AxC) \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma, x : A \vdash M : \sigma}\ (w)$$

$$\frac{\Gamma, x : \sigma \vdash M : A}{\Gamma \vdash \lambda x.M : \sigma \multimap A}\ (\multimap I) \qquad \frac{\Gamma \vdash M : \sigma \multimap A \qquad \Delta \vdash N : \sigma \quad \Gamma\#\Delta}{\Gamma, \Delta \vdash MN : A}\ (\multimap E)$$

$$\frac{\Gamma, x_1 : \xi, \ldots, x_n : \xi \vdash M : \tau \quad \xi \in \mathcal{T}_\sigma \cup \mathcal{T}_\mathfrak{s}}{\Gamma, x : !\xi \vdash M[x/x_1, \ldots, x/x_n] : \tau}\ (m) \qquad \frac{\Gamma \vdash M : \sigma}{!\Gamma \vdash M : !\sigma}\ (sp)$$

$$\frac{\Gamma \vdash M_1 : !^iB \qquad \Delta, x : !^i\mathrm{Clos}(\Gamma, \Delta; B) \vdash M_2 : A \quad \Gamma\#\Delta}{\Gamma, \Delta \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2 : A}\ (let)$$

Figure 1 The typing rules of MLSTA.

3 Gentle introduction to MLSTA

The main feature of soft type systems is their ability to control multiplication of data. One piece of the mechanism is realised by the (m) rule. This rule makes explicit the demand of an operator to duplicate some portion of data. The multiplication is reflected by ! in the type. The second piece of the mechanism is realised by the (sp) rule. The latter is used when the term it types is to be directly multiplied by a substitution present in the β-reduction. Note that when a term M_1 is directly multiplied k_1 times and is used inside another term M_2 that is directly multiplied k_2 times the number of occurrences of M_1 at some point of the computation may be as high as $k_1 \cdot k_2$.

The traditional soft type assignment systems use the full polymorphism of the System F. This makes possible to conveniently define data types and iterators over them, but it leads to undecidability of the type inference and type checking problems [6]. In our proposal, we provide access to the polymorphic expressibility in a structured fashion. It is achieved through two mechanisms. The first one brings a few fixed algebraic types: strings, naturals, booleans and products with appropriate constructors, destructors and iterators. The set of algebraic types could be richer. This, however, would make the design of the model language unnecessarily complicated. The second mechanism is the traditional let-polymorphism which makes possible to define generic operations that work for different kinds of data.

The language we propose contains also a type of lists of booleans $\mathbb{S}_i^{!j}$, called here strings, which is our main recursive data structure The numbers i and j describe the complexity of the

$$\texttt{let}\ x = M_1\ \texttt{in}\ M_2 \to_\beta M_2[M_1/x]$$
$$(\lambda x.M)N \to_\beta M[N/x]$$

Constants:

The typed constants $c : \tau$ listed below belong to the set $\mathrm{Const}_{\mathrm{MLSTA}}$.

Product

$$\begin{array}{rcl}
\langle\cdot,\cdot\rangle & : & \forall\alpha\beta.\alpha \multimap \beta \multimap \alpha\otimes\beta \\
\texttt{match} & : & \forall\alpha\beta\gamma.\alpha\otimes\beta \multimap (\alpha\multimap\beta\multimap\gamma)\multimap\gamma \\
 & & \texttt{match}\ \langle M_1, M_2\rangle\ N \to_\delta N\ M_1\ M_2
\end{array}$$

Booleans

$$\begin{array}{rcl}
\mathbf{0} & : & \mathbb{B} \\
\mathbf{1} & : & \mathbb{B} \\
\texttt{ifte} & : & \forall\alpha.\mathbb{B}\multimap\alpha\multimap\alpha \\
 & & \texttt{ifte}\ \mathbf{0}\ M_1\ M_2 \to_\delta M_1 \\
 & & \texttt{ifte}\ \mathbf{1}\ M_1\ M_2 \to_\delta M_2
\end{array}$$

Natural numbers

$$\begin{array}{rcl}
\underline{n} & : & \mathbb{N}^{!j} \\
\texttt{add} & : & \mathbb{N}^{!j_1}\multimap\mathbb{N}^{!j_2}\multimap\mathbb{N}^{!\max(j_1,j_2)+1} \\
 & & \texttt{add}\ \underline{n}\ \underline{m} \to_\delta \underline{n+m} \\
\texttt{mul} & : & \mathbb{N}^{!j_1}\multimap\mathbb{N}^{!j_2}\multimap\mathbb{N}^{!(j_1+j_2)} \\
 & & \texttt{mul}\ \underline{n}\ \underline{m} \to_\delta \underline{n*m} \\
\texttt{iter} & : & \forall\alpha.\mathbb{N}^{!j}\multimap !^j(\alpha\multimap\alpha)\multimap\alpha\multimap\alpha \\
 & & \texttt{iter}\ \underline{n}\ F\ M \to_\delta F\ (\dots(F\ M)\dots) \quad (F \text{ applied } n \text{ times to } M)
\end{array}$$

Strings

$$\begin{array}{rcl}
[\cdot;\dots;\cdot] & : & \mathbb{B}^i \multimap\cdots\multimap\mathbb{B}^i\multimap\mathbb{S}_i^{!j} \\
\texttt{create} & : & \mathbb{N}^{!j}\multimap !^j\mathbb{B}^i\multimap\mathbb{S}_i^{!j} \\
 & & \texttt{create}\ \underline{n}\ M \to_\delta [M;\dots;M] \quad (n \text{ copies of } M) \\
\texttt{concat} & : & \mathbb{S}_i^{!j_1}\multimap\mathbb{S}_i^{!j_2}\multimap\mathbb{S}^{!\max(j_1,j_2)+1} \\
 & & \texttt{concat}\ [M_1;\dots;M_m]\ [N_1;\dots;N_n] \to_\delta [M_1;\dots;M_m;N_1;\dots;N_n] \\
\texttt{len} & : & \mathbb{S}_i^{!j}\multimap\mathbb{N}^{!j} \\
 & & \texttt{len}\ [M_1;\dots;M_m] \to_\delta \underline{m}
\end{array}$$

Looping constructs

$$\begin{array}{rcl}
\texttt{localvars} & : & \forall\alpha.\mathbb{S}_i^{!j}\multimap\cdots\multimap\mathbb{S}_i^{!j}\multimap\alpha\multimap(\mathbb{S}_i^{!j})^k\boxtimes\alpha \quad (k+1 \text{ arguments}) \\
\mathfrak{p}_0,\dots,\mathfrak{p}_{k-1} & : & \forall\alpha.(\mathbb{S}_i^{!j})^k\boxtimes\alpha\multimap\mathbb{S}_i^{!j+1} \\
\mathfrak{p}_k & : & \forall\alpha.(\mathbb{S}_i^{!j})^k\boxtimes\alpha\multimap\alpha \\
 & & \mathfrak{p}_l(\texttt{localvars}\ M_0\dots\ M_k) \to_\delta M_l \\
\texttt{step} & : & ((\mathbb{B}^i\otimes\mathbb{B})^k\otimes\mathbb{B}^q\multimap(\mathbb{B}^i\otimes\mathbb{B}^l)^k\otimes\mathbb{B}^q)\multimap(\mathbb{S}_i^{!j})^k\boxtimes\mathbb{B}^q\multimap(\mathbb{S}_i^{!j})^k\boxtimes\mathbb{B}^q \\
 & & \quad \text{where } l = \lceil\log(k+1)\rceil \\
 & & \texttt{step}\ F\ (\texttt{localvars}\ M_0\dots\ M_k) \to_\delta \texttt{localvars}\ M_0'\dots\ M_k' \\
 & & \quad \text{where } F\ \langle \mathrm{hd}(M_0),\dots,\mathrm{hd}(M_{k-1}),M_k\rangle \to_{\beta\delta}^* \langle N_0,\dots,N_{k-1},M_k'\rangle \\
 & & \qquad N_i = \langle P_i, \overline{l_i}\rangle \\
 & & \qquad\quad \text{where } \overline{l} \text{ denotes the binary encoding of a number } l \\
 & & \qquad M_j' = P_{i_1} :: \cdots :: P_{i_w} :: \mathrm{tl}(M_j) \\
 & & \qquad\quad \text{where } i_1\dots i_w \text{ is the subsequence of } 0\dots k-1 \text{ such that} \\
 & & \qquad\quad \forall p\ l_{i_p} = j \text{ and } M_{i_p} \neq [\,] \\
 & & \qquad \mathrm{hd}([\,]) = \langle\mathbf{0}^i,\mathbf{1}\rangle \qquad \mathrm{tl}([\,]) = [\,] \\
 & & \qquad \mathrm{hd}([A_1;\dots;A_m]) = \langle A_1,\mathbf{0}\rangle \qquad \mathrm{tl}([A_1;\dots;A_m]) = [A_2;\dots;A_m]
\end{array}$$

Figure 2 MLSTA constants, their types and reduction rules of MLSTA. Note that $j, j_1, j_2 \geq 1$ when they represent the number of !'s. In addition the superscript of the form $!j$ indicates that the type has j 'hidden' bangs (they become explicit after a translation to STA, see Fig. 4).

string: i corresponds to the size of a symbol in the string — each symbol is of type $\mathbb{B}^i$, which is a shorthand for $\mathbb{B} \otimes \cdots \otimes \mathbb{B}$ (i times), while j, roughly speaking, reflects the complexity of the string creation. Strings are used in the framework to simulate PTIME computations. We provide a number of usual operators over strings. A string which combines n pieces of basic data can be obtained using the bracket construct $[a_1; \ldots; a_n]$. We can also create a string of n copies of a particular piece of data `create` n a. The strings s_1, s_2 obtained in one or another way can be combined by concatenation done with `concat` s_1 s_2.

Note that traditional iterative operation `fold` from functional languages is missing in MLSTA. In principle we could introduce it to our language. However the traditional type of the operation imposes too strict restrictions on the type of function that operates on strings. In particular it is impossible to transform one string to another since this requires duplication of the string constructor which is prohibited in the context of linear types.

We adopt a different approach. In order to do iterative programs on strings, we introduce a `step` function, inspired by the encoding of a Turing Machine in [24]. Its functionality is to put a program fragment, represented by the first argument f, into the context of an iterative loop. Then `step` takes a tuple of several strings $(\mathbb{S}_i^{!j})^k \boxtimes \alpha$ "federated" into a context of "local variables". This structure is created with the `localvars` operation and it is the structure over which the iteration is actually performed. In one iteration step the heads of the strings can be freely moved around or dumped, but not duplicated. Some information can also be stored in the accumulator, represented here by α. One application of `step` f maps the operation on heads done by its argument function f onto the corresponding operation on federated strings. The mapped operation can then be iterated by `iter` as many times as necessary in order to complete the whole string processing and in the end we can extract the result using one of the projections $\mathfrak{p}_0, \ldots, \mathfrak{p}_k$. Note that since the calculation is done in constant space, all allocations must occur before starting the iteration. Indeed, it would be impossible to extend any of the strings in the iterated function, as this would make types incompatible. The simplest example of `step` usage is string reversal:

```
let rev = let fr = λ ((a₀,b₀),(a₁,b₁),q). ((a₀,1̅),(a₁,1̅),q) in
  λs. 𝔭₁ (iter (len s) (step fr) (localvars s [] 𝟎))
```

In the above example we use a few syntactic simplifications, which are straightforward to translate into core MLSTA. The federated tuple consists here of two strings and a (dummy) boolean. In the beginning, the left string is the initial string s and the right one is the empty string. The function `fr` takes two "heads": a_0 and a_1 paired with the booleans b_0 and b_1 respectively, carrying the information if the given head is real or dummy (in case the given string is empty). Its result tells the `step` function to attach both a_0 and a_1 to the right string in the order from right to left, i.e., a_1 is attached first (where it came from) and then a_0. In case one of the strings was initially empty (here it is only possible for the right one), the head on the corresponding position would be attached with a dummy cons (observe condition $M_{i_p} \neq []$ in Fig. 2), i.e. would not be attached at all.

Now it is also possible to write the map function on strings:

```
let map = let fm = λ f ((a₁,b₁), (ar,br), (a₂,b₂)), q).
    if q then ((a₁,1̅), (ar,1̅), (a₂,2̅), if b₁ then 𝟎 else 𝟏)
    else ((a₁,1̅), (f ar,2̅), (a₂,2̅), 𝟏)
  in λ f s. let n = len s in
    𝔭₂ (iter (add (mul 2̲ n) 1̲) (step (fm f)) (localvars s [] [] 𝟎));;
```

Another syntactic simplification can be seen here: although the two branches of `if` share variables, the term can be written as linear using the trick `if b then t[a] else s[a]` $\equiv$

`(if b then λa.t[a] else λa.s[a]) a`. The function operates in two phases: in the first phase the input string (put initially on the left federated string) is reversed and placed on the middle string. In the second phase the middle string is mapped using f to the right string, and reversed back to the original order in the process. The number of iterations is $2n+1$, which is n for each phase and 1 for phase change. The phase number is encoded as a boolean, initially true ($\mathbf{0}$) and then changed to false ($\mathbf{1}$).

```
let fsel = λ ((sortedPartHead, sortedPartHasHead),
              (maxSoFar, maxUsable),
              (unsortedPartHead, unsortedPartHasHead),
              (soFarSeenHead, soFarSeenHasHead), x).
  if not maxUsable and unsortedPartHasHead and not soFarSeenHasHead then
     (* start of selection phase *)
    (sortedPartHead, 0̄), (maxSoFar, 1̄),
    (unsortedPartHead, 1̄), (soFarSeenHead, 3̄), x
  else if maxUsable and unsortedPartHasHead then (* selection phase in progress *)
    cmp maxSoFar unsortedPartHead (fun smaller greater ->
      (sortedPartHead, 0̄), (greater, 1̄),
      (smaller, 3̄), (soFarSeenHead, 3̄), x )
  else if maxUsable and not unsortedPartHasHead then (* end of selection phase *)
    if sortedPartHasHead then (* end of selection phase for the first selection *)
      (maxSoFar, 0̄), (sortedPartHead, 0̄),
      (unsortedPartHead, 2̄), (soFarSeenHead, 3̄), x
    else (* end of selection phase for other selections *)
      (sortedPartHead, 0̄), (maxSoFar, 0̄),
      (unsortedPartHead, 2̄), (soFarSeenHead, 3̄), x
  else (* preparation for the next selection phase *)
    if unsortedPartHasHead and soFarSeenHasHead then
      (sortedPartHead, 0̄), (maxSoFar, 1̄),
      (soFarSeenHead, 2̄), (unsortedPartHead, 2̄), x
    else
      (sortedPartHead, 0̄), (maxSoFar, 1̄),
      (unsortedPartHead, 2̄), (soFarSeenHead, 2̄), x

let ssort = λl. let n = len l in
  p₀ (iter (add (mul n n) n) (step fsel) (localvars [] [] l [] 0))
```

Figure 3 Selection sort. We encourage the reader to try to understand the algorithm herself.

It is very interesting to note that the type of a map function defined in this way is $!^{j+2}(\mathbb{B}^i \multimap \mathbb{B}^i) \multimap !\mathbb{S}_i^{!j} \multimap \mathbb{S}_i^{!j+1}$, while the type of a map function defined directly in STA corresponds to $!^j(\mathbb{B}^i \multimap \mathbb{B}^i) \multimap \mathbb{S}_i^{!j} \multimap \mathbb{S}_i^{!j+1}$. The difference comes from the fact that in MLSTA one iterates over natural numbers and in STA directly on the string itself.

Using this technique it is possible to program more complex functions on lists, e.g. sorting, in particular selection sort, as shown in Fig. 3. This example uses another syntactic trick: since boolean values can be freely multiplied using terms similar to $\mathbf{cnt} \equiv \lambda b.\texttt{ifte}\ b\ \langle \mathbf{0}, \mathbf{0} \rangle\ \langle \mathbf{1}, \mathbf{1} \rangle$ one does not need to worry about how many times a given boolean variable is used in the term. Technically, sorting consists in running n phases of selecting the largest element from unsorted remaining part of the initial string. In each phase one needs to reverse the list twice, that is why we need $n^2 + n$ steps. It is interesting to note that the choice of applying the cons in the same order as they appeared originally comes at a cost of breaking symmetry of certain operations between two strings. Indeed, while it is straightforward to reverse a string from left to right (as in the `rev` example above), reversing it from right to left (as is done in the third case in Fig. 3) is a bit more technical.

It is worth stressing that the simulation of a Turing Machine we present below is in fact a paradigmatic example of a natural computation that can be performed in our language.

4 Properties of MLSTA

Many of the results in this paper can be obtained in a simpler way when we operate not just on any derivation, but on a derivation in a special, regular form. We start with its presentation, which is of interest not only for technical reasons but also, as usual in such cases, it indicates the presence of a few important tautologies (however, their further exploration goes beyond the topic of the paper).

▶ **Definition 1** (derivations in normal form). A derivation $\mathfrak{D}$ of MLSTA is in *normal form* when $\mathfrak{D} = \langle\hat{\mathfrak{D}}, x : A\rangle^w$ with $\hat{\mathfrak{D}}$ in normal form and $x \notin \mathrm{FV}(\hat{\mathfrak{D}}_{term})$ or is in (m)-normal form.

A derivation $\mathfrak{D}$ of MLSTA is in (m)*-normal form* when $\mathfrak{D} = \langle \xi x : A.\langle\hat{\mathfrak{D}}, x : A\rangle^w, y : !^n A\rangle^{mn}$ with $\hat{\mathfrak{D}}$ in (m)-normal form and $x \notin \mathrm{FV}(\hat{\mathfrak{D}}_{term})$ or is in (sp)-normal form.

A derivation $\mathfrak{D}$ of MLSTA is in (sp)*-normal form* when $\mathfrak{D} = \langle\hat{\mathfrak{D}}\rangle^{sp}$ with $\hat{\mathfrak{D}}$ in (sp)-normal form or in logical normal form.

A derivation $\mathfrak{D}$ of MLSTA is in *logical normal form* when it is

- x^A for some variable x,
- $x^{A\leq\mathfrak{s}}$ for some variable x,
- $\lambda x^\sigma.\hat{\mathfrak{D}}$ for some variable x and $\hat{\mathfrak{D}}$ in logical normal form,
- $\lambda x^\sigma.\langle\hat{\mathfrak{D}}, x : A\rangle^w$ for some variable x and $\hat{\mathfrak{D}}$ in logical normal form with $x \notin \mathrm{FV}(\hat{\mathfrak{D}}_{term})$,
- $\lambda x^\sigma.\langle \xi x : A.\hat{\mathfrak{D}}, x : !^k A\rangle^{mk}$ for some variable x and $\hat{\mathfrak{D}}$ in logical normal form with $x \notin \mathrm{FV}(\hat{\mathfrak{D}}_{term})$,
- $\mathfrak{D}_1\mathfrak{D}_2$ where $\mathfrak{D}_1$ is in logical normal form and $\mathfrak{D}_2$ is in (sp)-normal form,
- $\mathtt{let}\ x^{!^i\forall\vec{\alpha}.B} = \mathfrak{D}_1\ \mathtt{in}\ \mathfrak{D}_2$ where $\mathfrak{D}_1$ is in normal form and $\mathfrak{D}_2$ is in logical normal form.

▶ Proposition 2 (properties of derivations). If $\Gamma \vdash_{\mathrm{MLSTA}} M : \sigma$ then the judgement has a derivation in normal form.

Proof. The proof is using a special kind of reduction the normal forms of which are the defined above normal forms. ◀

As a corollary we obtain a condition that says in which way we can drop a bang in the final type of a term.

▶ **Corollary 3** (dropping final bang). *If* $\Gamma \vdash_{\mathrm{MLSTA}} M : !\sigma$ *then there is a context* Γ' *such that* $!\Gamma' \subseteq \Gamma$ *and* $\Gamma' \vdash_{\mathrm{MLSTA}} M : \sigma$ *and for each* $x : \tau \in \Gamma \backslash !\Gamma'$ *we have* $x \notin \mathrm{FV}(M)$*.*

Proof. By Prop. 2 there is a derivation of $\Gamma \vdash_{\mathrm{MLSTA}} M : !\sigma$ in normal form. We observe that we can one by one remove the final (w) and (m) rules. At the end we have to arrive at an (sp) rule since no logical normal form can assign a ! type to a term. ◀

A crucial part of the subject reduction proof is the interaction between substitutions and the derivations. This is expressed in the following proposition.

▶ Proposition 4 (derivations and substitutions). If $\Gamma \vdash M : A$ then for each substitution $U : \mathcal{V} \rightharpoonup \mathcal{T}_A$ we have $U(\Gamma) \vdash M : U(A)$.

Proof. Induction over the inference of $\Gamma \vdash M : A$ by cases according to its final rule. ◀

Let expressions
$\mathtt{let}\ x = M\ \mathtt{in}\ N = (\lambda x.M)N$

Tensor products
$A \otimes B = \forall\alpha.(A \multimap B \multimap \alpha) \multimap \alpha$
$\langle\cdot,\cdot\rangle = \lambda mnz.zmn$
$\mathtt{match} = \lambda pf.pf$

Booleans
$\mathbb{B} = \forall\alpha.\alpha \multimap \alpha \multimap \alpha$
$\mathbf{0} = \lambda tf.t$
$\mathbf{1} = \lambda tf.f$
$\mathtt{ifte} = \lambda bxy.bxy$

Natural numbers
$\mathbb{N}^{!j} = \forall\alpha.!^j(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$
$\underline{n} = \lambda fx.f(\dots(fx)\dots)$ (f applied n times to x)
$\mathtt{add} = \lambda pqsz.ps(qsz)$
$\mathtt{mul} = \lambda pqsz.p(qs)z$
$\mathtt{iter} = \lambda pfx.pfx$

Strings
$\mathbb{S}^{!j}_i = \forall\alpha.!^j(\mathbb{B}^i \multimap \alpha \multimap \alpha) \multimap \alpha \multimap \alpha$
$[\cdot;\dots;\cdot] = \lambda a_1 \dots a_n cz.ca_1(\dots(ca_n z)\dots)$
$\mathtt{create} = \lambda nacz.n(ca)z$
$\mathtt{concat} = \lambda s_1 s_2 cz.s_1 c(s_2 cz)$
$\mathtt{len} = \lambda sfz.s(\lambda x.f)z$

Local variables (for the sake of clarity we retain abbreviations related to $\otimes$ and $\mathbb{B}$)
$(\mathbb{S}^{!j}_i)^k \boxtimes A = \forall\alpha.!^{j+1}(\mathbb{B}^i \multimap \alpha \multimap \alpha) \multimap (\alpha \multimap \alpha)^k \otimes A$
$\mathtt{localvars} = \lambda s_0 \dots s_{k-1} q \lambda c.\langle s_0 c, \dots, s_{k-1}c, q\rangle$
$\mathtt{p}_n = \lambda v.\lambda c.\mathtt{match}\ (vc)\ \lambda s_0 \dots s_{k-1} q.s_i$ for $n = 0 \dots k-1$
$\mathtt{p}_k = \lambda v.\mathtt{match}\ (v\lambda x.I)\ \lambda s_0 \dots s_{k-1} q.q$
$\mathtt{step} =$
$\quad \lambda fvc.\mathtt{match}\ (\mathsf{Dec}\ vc)\ \lambda c_0 a_0 b_0 t_0 \dots c_{k-1} a_{k-1} b_{k-1} t_{k-1} q.$
$\qquad \mathsf{Enc}\ c_0 \dots c_{k-1}(f\langle\langle a_0, b_0\rangle, \dots, \langle a_{k-1}, b_{k-1}\rangle, q\rangle)\langle t_0, \dots, t_{k-1}\rangle$, where
$\mathsf{Dec}\ = \lambda vc.\mathtt{match}\ (v\ \mathtt{F}[c])\ \lambda\tilde{s}_0 \dots \tilde{s}_{k-1} q.$
$\quad \mathtt{match}\ (\tilde{s}_0\langle\lambda az.z, \mathbf{0}^i, \mathbf{1}, \lambda z.z\rangle)\ \lambda c_0 a_0 b_0 t_0.\ \dots$
$\qquad \mathtt{match}\ (\tilde{s}_{k-1}\langle\lambda az.z, \mathbf{0}^i, \mathbf{1}, \lambda z.z\rangle)\ \lambda c_{k-1} a_{k-1} b_{k-1} t_{k-1}.$
$\qquad\quad \langle c_0, a_0, b_0, t_0, \dots, c_{k-1}, a_{k-1}, b_{k-1}, t_{k-1}, q\rangle$
where $\mathtt{F}[c] = \lambda az.\mathtt{match}\ z\ \lambda c'a'b't'.\langle c, a, \mathbf{0}, c'a' \circ t'\rangle$
and
$\mathsf{Enc}\ = \lambda c_0 \dots c_{k-1} wv.\mathtt{match}\ w\ \lambda h_0 \dots h_{k-1} q'.$
$\quad \mathtt{match}\ h_0\ \lambda a'_0 p_0.\ \dots\ \mathtt{match}\ h_{k-1}\ \lambda a'_{k-1} p'_{k-1}.$
$\qquad \mathtt{match}\ (\mathsf{Add}\ p_0 c_0 a'_0(\dots(\mathsf{Add}\ p_{k-1} c_{k-1} a'_{k-1} v)\dots))$
$\qquad\quad \lambda s'_0 \dots s'_{k-1}.\langle s'_0, \dots, s'_{k-1}, q'\rangle$, where
$\mathsf{Add}\ = \lambda p.\mathtt{match}\ p\ \mathtt{CASE}_{\lceil\log(k+1)\rceil}[I_0, \dots, I_{k-1}][I]$, where
$I_n = \lambda cam.\mathtt{match}\ m\ \lambda s_0 \dots s_{k-1}.\langle s_0, \dots, s_{n-1}, c\,a \circ s_n, s_{n+1} \dots, s_{k-1}\rangle$, and
$I = \lambda cam.m$, and

$$\mathtt{CASE}_w[t_0,\dots,t_{n-1}][t] = \begin{cases} \lambda b_0.\mathtt{ifte}\ b_0\ \mathtt{CASE}_{w-1}[t_0,\dots,t_{2^{w-1}-1}][t] & \\ \qquad\qquad \mathtt{CASE}_{w-1}[t_{2^{w-1}},\dots,t_{n-1}][t] & \text{if } w > 0 \text{ and } n > 2^{w-1} \\ \lambda b_0.\mathtt{ifte}\ b_0\ \mathtt{CASE}_{w-1}[t_0,\dots,t_{n-1}][t] & \\ \qquad\qquad \mathtt{CASE}_{w-1}[\,][t] & \text{if } w > 0 \text{ and } 0 < n \le 2^{w-1} \\ t_0 & \text{if } w = 0 \text{ and } n = 1 \\ \lambda b_0 \dots b_{w-1}.t & \text{if } n = 0 \end{cases}$$

Figure 4 Translation of MLSTA to STA.

The proposition above makes it possible to describe the way the type instantiation operation works in the context of derivations.

▶ Proposition 5. If $\Delta \vdash N : A$ then $\Delta \vdash N : A'$ if $\mathrm{Clos}(\Delta, A) \ge A'$.

Proof. This is an instance of Prop. 4, since if $A' = U(A)$ then $\mathrm{dom}(U) \cap \mathrm{FTV}(\Delta) = \emptyset$ and therefore $U(\Delta) = \Delta$. ◀

We can now combine the previous two statements and obtain the substitution lemma for our system.

▶ **Lemma 6** (substitution lemma). *If* $\Gamma; x : !^i\forall\overline{\alpha}.A \vdash_{\mathrm{MLSTA}} M : \tau$ *and* $\Delta \vdash_{\mathrm{MLSTA}} N : !^i A$ *where* $\overline{\alpha} \notin \mathrm{FTV}(\Delta)$*, then* $\Gamma; \Delta \vdash_{\mathrm{MLSTA}} M[N/x] : \tau$

Proof. The proof can be done almost in the same way as the proof of the Substitution Lemma 2.7 in [13], i.e., by generalising the statement to many simultaneous substitutions and proceeding by induction on the derivation by analysis of the last rule. The new/different rules in MLSTA are (Ax), (AxC) and (let).

If the last step is (Ax) then $M = x$ and we have $x : \forall\overline{\alpha}.A \vdash x : B$ with $\forall\overline{\alpha}.A \geq B$ and $\Delta \vdash N : A$ where $\overline{\alpha} \notin \mathrm{FTV}(\Delta)$. Therefore one has $\Delta \vdash N : B$, by Prop. 5, because $\mathrm{Clos}(\Delta; A) \geq B$.

It is impossible that the last step is (AxC), because the context is empty.

If the last rule is (let), the result follows easily by induction hypothesis.

Other MLSTA rules are identical to their $\mathrm{STA_B}$ counterparts. ◀

As a result of the substitution lemma we obtain the subject reduction property.

▶ **Theorem 7** (subject reduction). *If $\Gamma \vdash_{\mathrm{MLSTA}} M : A$ and $M \to_{\beta\delta} M'$ then $\Gamma \vdash_{\mathrm{MLSTA}} M' : A$.*

5 MLSTA and PTIME

Observe that the MLSTA can easily be embedded into STA, in the same fashion as usual ML can be embedded in System F [17, Section 3], see Fig. 4. This gives us polynomial guarantee on the length of reductions.

▶ **Theorem 8.** *Given a derivation $\Gamma \vdash_{\mathrm{MLSTA}} M : \sigma$, the number of reductions from M can be bounded by $|M|^{O(d)}$ where: $|M|$ is the* size *of the term M, defined as usual with one caveat — the size of a natural constant $\underline{n}$ is n; d is the* degree *of a derivation, defined as the maximum nesting of (sp) rules in the derivation.*

Proof. The translation of MLSTA into STA preserves types and degree of derivations, and guarantees that every MLSTA reduction step is translated to a number of steps in STA. The result of translation is bigger only by a linear factor from its original. In the end the polynomial bound established for STA (Theorem 15 in [14]) works for MLSTA as well. ◀

Now, we aim at a proof that each TM in PTIME can be simulated in MLSTA by a term. We start by a version of Lemma 16 and 17 from [14] for our built-in naturals and booleans:

▶ **Lemma 9** (polynomials). *Let P be a polynomial with positive coefficients in the variable x of the degree $\deg(P)$. There is a term $\underline{P}$ such that $\vdash_{\mathrm{MLSTA}} \underline{P} : !^{\deg(P)}\mathbb{N}^{!1} \multimap \mathbb{N}^{!(2\deg(P)+1)}$.*

▶ **Lemma 10** (boolean functions). *Each boolean total function $f : \{0,1\}^n \to \{0,1\}^m$ for $m, n \geq 1$ can be defined by a term $\underline{f}$ typable in MLSTA as $\vdash \underline{f} : \mathbb{B}^n \multimap \mathbb{B}^m$.*

The following theorem shows how one can encode polynomial Turing Machines in MLSTA. For simplicity, we encode only deterministic machines which move their head (left or right) at every step. Therefore a transition function can be encoded as $\delta : \mathbf{\Sigma} \times \mathbf{Q} \to \mathbf{\Sigma} \times \mathbf{Q} \times \mathbb{B}$, where $\mathbf{\Sigma}$ is the alphabet, $\mathbf{Q}$ the set of states and the last boolean value denotes the head move: $\mathbf{0}$ denotes 'left' and $\mathbf{1}$ 'right'. Since $\mathbf{\Sigma}$ and $\mathbf{Q}$ are finite, there exist sufficiently large k and k', such that $\mathbf{\Sigma} \equiv \mathbb{B}^k$ with 0^k representing blank and $\mathbf{Q} \equiv \mathbb{B}^{k'}$. Hence, according to Lemma 10, there exists $\underline{\delta} : \mathbb{B}^k \otimes \mathbb{B}^{k'} \multimap \mathbb{B}^k \otimes \mathbb{B}^{k'} \otimes \mathbb{B}$ representing δ.

▶ **Theorem 11.** *Let $\mathcal{M}$ be a Turing Machine. There is an MLSTA term $M_{\mathcal{M}}$ such that $\vdash_{\mathrm{MLSTA}} M_{\mathcal{M}} : !^d\mathbb{S}_k \multimap \mathbb{B}$ for some d where for each input s the term $M_{\mathcal{M}}\ \underline{s}$ reduces to $\mathbf{0}$ using $R(|s|)$ of reductions with R being a polynomial of degree $O(d)$ if and only if $\mathcal{M}$ accepts s. Moreover, $M_{\mathcal{M}}$ can be constructed from $\mathcal{M}$ in polynomial time.*

Proof. Let $\mathcal{M}$ be a deterministic TM with alphabet $\mathbf{\Sigma} \equiv \mathbb{B}^k$, the set of states $\mathbf{Q} \equiv \mathbb{B}^{k'}$ and transition function δ. Let S be a polynomial defining the maximal length of auxiliary tape of $\mathcal{M}$ for all input strings of a given length. Let T be a polynomial defining the maximal number of steps needed for all input strings of a given length. It is enough to construct the space using $\underline{S}$, the time using $\underline{T}$ and combine it all into the term $M_{\mathcal{M}}$ equal

$$\begin{aligned}&\lambda s.\mathtt{let}\ \mathit{time} = \underline{T}\ (\mathtt{len}\ s)\ \mathtt{in}\\ &\quad \mathtt{let}\ \mathit{tape}_0 = \mathtt{concat}\ s\ (\mathtt{create}\ (\underline{S}\ (\mathtt{len}\ s))\ \mathbf{0}^k)\ \mathtt{in}\\ &\quad \mathtt{let}\ \mathit{conf}_0 = \mathtt{localvars}\ [\,]\ \mathit{tape}_0\ \underline{q_0}\ \mathtt{in}\\ &\quad \mathit{is\_acc}\ (\mathfrak{p}_2\ (\mathtt{iter}\ \mathit{time}\ (\mathtt{step}\ F_\delta)\ \mathit{conf}_0))\end{aligned}$$

where $\mathit{is\_acc}$ is a function of type $\mathbf{Q} \multimap \mathbb{B}$ returning $\mathbf{0}$ if the state it receives as input is accepting and F_δ is a simple wrapper around $\underline{\delta}$ to match the input specification of `step`.

$$\begin{aligned}F_\delta = \lambda&\mathtt{((al,bl),(a,b),q).\ match\ (}\underline{\delta}\mathtt{(a,q))\ }\lambda\mathtt{a'q'd.\ ifte\ d\ ((al,}\overline{1}\mathtt{),(a',}\overline{1}\mathtt{),q')}\\ &\mathtt{ifte\ bl\ ((a',}\overline{0}\mathtt{),(al,}\overline{0}\mathtt{),q')\ ((al,}\overline{0}\mathtt{),(a',}\overline{0}\mathtt{),q')}\end{aligned}$$

The degree of the type derivation of $M_{\mathcal{M}}$, the degree of terms $M_{\mathcal{M}}\ \underline{s}$ for any s and the parameter d depend in a linear way on the degree of the polynomials T and S. By Theorem 8 each term $M_{\mathcal{M}}\ \underline{s}$ can be reduced to a normal form in the number of reductions bound by a polynomial of degree $O(d)$. ◀

6 Decidability of typechecking with ML polymorphism

MLSTA enjoys decidable typechecking, type inference and typability problems. To prove this we adapt the algorithm W also known as Hindley-Milner algorithm following [29].

The *type checking problem* (TCP) is the problem: given a term M, a type A, and a context Γ, is $\Gamma \vdash M : A$ derivable? The *type inference problem* (TIP) is the problem: given a term M and a context Γ, is there a type A such that $\Gamma \vdash M : A$ is derivable? Finally, the *typability problem* (TP) is the problem: given a term M, are there a context Γ and a type A such that $\Gamma \vdash M : A$ is derivable? We describe the way the problems can be solved with W-lin in the proof of Theorem 16.

The basic building block of the algorithm is the procedure of unification [28]. To make use of the procedure we divide the type variables $\mathcal{V}$ into two infinite disjoint parts $\mathcal{V}_v \cup \mathcal{V}_c = \mathcal{V}$. The set $\mathcal{V}_v$ contains *substitutable type variables*, called simply type variables below, and $\mathcal{V}_c$ contains variables that serve the role of constants in unification, called constants below. A substitution U that substitutes expressions on type variables (e.g. α, β etc.) is a unifier of $A \doteq A'$ when $U(A) = U(A')$. It is important to note that elements of $\mathcal{V}_v$ do not occur in the substitution applied to obtain an instance of a type in rules (Ax) and (AxC) in Fig. 1. This unification enjoys the most general unifier property, but we cannot use it directly here. Therefore we provide a special version of the most general unifier in Def. 13 below.

To express the procedure for typechecking, type inference, or typability we need a few technical definitions. We say that Γ is *full* with regard to a term M when $\mathrm{dom}(\Gamma) = \mathrm{FV}(M)$ this fact is denoted by $\mathrm{Full}(\Gamma; M)$. We say that Γ is *linear* with regard to a term M when for each $x : \sigma \in \Gamma$ the variable x occurs freely exactly once in M. This is denoted by $\mathrm{Linear}(\Gamma; M)$. The set of algebraic constants defined in Fig. 1 is denoted as $\mathrm{Const}_{\mathrm{MLSTA}}$. The algorithm we study here is presented in Fig. 5. The input for the algorithm is an environment Γ, a term M, and a type A. The output is a substitution U and a set of equations E. The situation that U, E are a valid output for Γ, M, A is denoted as $\Gamma \vdash_{\mathrm{Wl}} M : A \rightsquigarrow (U, E)$.

$$\frac{\beta_0,\dots,\beta_n \text{ are fresh} \qquad c:\forall\alpha_1\cdots\alpha_n.A \in \mathrm{Const}_{\mathrm{MLSTA}}}{\vdash c:\emptyset, N\oslash A[\beta_1/\alpha_1,\dots,\beta_n/\alpha_n] \rightsquigarrow (\emptyset,\{N \doteq 0\})}\ (AxC)$$

$$\frac{}{x:N\oslash A \vdash x:N\oslash A \rightsquigarrow (\emptyset,\{N\doteq 0\})}\ (Ax) \qquad \frac{\Gamma\vdash M:N_2\oslash A_2 \rightsquigarrow (U,E) \quad x\notin \mathrm{FV}(M)}{\Gamma, x:N_1\oslash A_1 \vdash M:N_2\oslash A_2\rightsquigarrow (U,E)}\ (w)$$

$$\frac{\begin{array}{c}\Gamma, x:\alpha_1\oslash\alpha_2 \vdash M:\alpha_3\oslash\alpha_4 \rightsquigarrow (U_1,E_1) \qquad \alpha_0,\alpha_1,\alpha_2,\alpha_3,\alpha_4 \text{ are fresh}\\ \mathsf{mgu}(\{U_1(\alpha_0\oslash((\alpha_1\oslash\alpha_2)\multimap\alpha_4))\doteq U_1(N\oslash A)\}) = (U_2,E_2)\\ E_2' = E_1\cup E_2\cup\{N\doteq 0, \alpha_3\doteq 0\} \qquad \mathrm{Full}(\Gamma;\lambda x.M) \quad \mathrm{Linear}(\Gamma;\lambda x.M)\end{array}}{\Gamma\vdash \lambda x.M : N\oslash A \rightsquigarrow (U_2\circ U_1, E_2')}\ (\multimap I)$$

$$\frac{\begin{array}{c}\Gamma\vdash M:\alpha_1\oslash((\alpha_2\oslash\alpha_3)\multimap N\oslash A)\rightsquigarrow (U_1,E_1) \qquad \alpha_1,\alpha_2,\alpha_3,\alpha_2',\alpha_3' \text{ are fresh}\\ \{x:\alpha_x'\oslash\alpha_x \mid x:N_x\oslash A_x\in\Delta, \alpha_x',\alpha_x \text{ are fresh}\}\vdash M':\alpha_2'\oslash\alpha_3'\rightsquigarrow (U_2,E_2)\\ \mathsf{mgu}(\{U_1(A_x)\doteq U_2(\alpha_x)\mid x:A_x\in\Delta\}\cup\{U_1(\alpha_2\oslash\alpha_3)\doteq U_2(\alpha_2'\oslash\alpha_3')\}) = (U_3,E_3)\\ E_4 = E_1\cup E_2\cup E_3 \cup\\ \{N\doteq 0, \alpha_1\doteq 0\}\cup\{N_x\doteq\alpha_2'+\beta_x\mid x:N_x\oslash A_x\in\Delta, \beta_x \text{ are fresh}\}\\ \Gamma\#\Delta \qquad \mathrm{Full}(\Gamma,\Delta;MM') \qquad \mathrm{Linear}(\Gamma,\Delta;MM') \qquad \mathrm{Full}(\Gamma;M) \qquad \mathrm{Full}(\Delta;M')\end{array}}{\Gamma,\Delta\vdash MM':N\oslash A\rightsquigarrow (U_3\circ U_2\circ U_1, E_4)}\ (\multimap E)$$

$$\frac{\begin{array}{c}\Gamma, x_1:\alpha\oslash\alpha',\dots,x_n:\alpha\oslash\alpha'\vdash M:\alpha_2\oslash\alpha_2'\rightsquigarrow (U_1,E_1)\\ \mathsf{mgu}(U_1(\alpha''\oslash\alpha')\doteq U_1(N_1\oslash A_1), U_1(\alpha_2\oslash\alpha_2')\doteq U_1(N_2\oslash A_2)) = (U_2,E_2)\\ E_3 = E_1\cup E_2\cup\{\alpha+1\doteq N_1\} \qquad \alpha,\alpha',\alpha_2,\alpha_2',\alpha'' \text{ are fresh}\\ \mathrm{Full}(\Gamma, x:N_1\oslash A_1; M[x/x_1,\dots,x/x_n]) \quad \mathrm{Full}(\Gamma, x_1:\alpha\oslash\alpha',\dots,x_n:\alpha\oslash\alpha'; M)\end{array}}{\Gamma, x:N_1\oslash A_1\vdash M[x/x_1,\dots,x/x_n]:N_2\oslash A_2\rightsquigarrow (U_2\circ U_1, E_3)}\ (m)$$

$$\frac{\begin{array}{c}\Gamma\vdash M_1:\alpha_1\oslash\alpha_2\rightsquigarrow (U_1,E_1) \qquad \alpha_1,\alpha_2,\beta_1\dots,\beta_n \quad \text{are fresh}\\ U_1(\Delta), x_1:\beta_1\oslash B_1,\dots,x_n:\beta_n\oslash B_n\vdash M_2':U_1(N'\oslash A)\rightsquigarrow (U_2,E_2)\\ M_2 = M_2'[x/x_1,\dots,x/x_n] \quad \mathsf{Fresh}(B_0,B_i) \text{ for } i=1,\dots,n \quad B_0 = \mathrm{Clos}(U_1(\Gamma);U_1(\alpha_2))\\ E_3 = E_1\cup E_2\cup\{\beta_i+\epsilon\doteq\alpha_1+\beta_i' \mid i=1,\dots,n, \beta_i' \text{ are fresh}\}\cup\\ \{N_x\doteq\alpha_1+\beta_x\mid x:N_x\oslash A_x\in\Gamma,\beta_x \text{ are fresh}\} \qquad \epsilon = [n>1]\\ \Gamma\#\Delta \quad \mathrm{Full}(\Gamma,\Delta;\mathtt{let}\ x=M_1\ \mathtt{in}\ M_2) \quad \mathrm{Linear}(\Gamma,\Delta;\mathtt{let}\ x=M_1\ \mathtt{in}\ M_2)\end{array}}{\Gamma,\Delta\vdash \mathtt{let}\ x=M_1\ \mathtt{in}\ M_2 : N\oslash A\rightsquigarrow (U_2\circ U_1,E_3)}\ (let)$$

Figure 5 The algorithm W-lin, ($[n>1]$ is 1 when $n>1$ and 0 otherwise).

The intent is that in case this relation holds then for each solution U' of E the relation $U\circ U'(\Gamma)\vdash_{\mathrm{MLSTA}} M : U\circ U'(A)$ holds as well. This property does not hold directly, but it is spelled out in full technical detail by Lemma 15(4).

The actual algorithm works on types in a different syntax defined by this grammar:

$$\begin{array}{l} A ::= N\oslash C \qquad N ::= \alpha \mid N_1+N_2 \mid n\\ C ::= \alpha \mid A_1\multimap A_2 \mid A_1\otimes A_2 \mid \mathbb{S}_i^{!j} \mid (\mathbb{S}_i^{!j})^k \boxtimes A \mid \mathbb{N}^{!j}\\ \mathfrak{s} ::= N\oslash\forall\vec{\alpha}.A\end{array}$$

where $n,i,j\in\mathbb{N}$. The set of types generated from the nonterminal A here is denoted as $\mathcal{T}_A^\oslash$, similarly generated from N is denoted as $\mathcal{T}_N^\oslash$, and from C — $\mathcal{T}_C^\oslash$, and the set of type schemes $\mathcal{T}_\mathfrak{s}^\oslash$. We use a general term $\oslash$-types to refer to elements of $\mathcal{T}_A^\oslash$. The elements generated from N are supposed to be expressions over natural numbers. We are free to perform any operations as soon as they are correct. For example the expression $3+4+\alpha$ is understood to be equal to $7+\alpha$. We divide the set of type variables $\mathcal{V}_v = \mathcal{V}_{vl}\cup\mathcal{V}_{vn}$ into two disjoint sets $\mathcal{V}_{vl}$ for type variables that are used to generate $\mathcal{T}_C^\oslash$ and $\mathcal{V}_{vn}$ for type variables that are use to generate $\mathcal{T}_N^\oslash$. We impose additional restriction on the substitutions below that variables in $\mathcal{V}_{vl}$ can be replaced by types from $\mathcal{T}_C^\oslash$ only and variables in $\mathcal{V}_{vn}$ by types from $\mathcal{T}_N^{\oslash}$ only. Types defined in (1) as $\mathcal{T}_\sigma$ can now be translated to $\mathcal{T}_A^{\oslash}$ and back using the following transformations:

▶ **Definition 12** (from types to $\oslash$-types and back)**.** We need a helper operation

We define now the transformation $(\cdot)^\bullet : \mathcal{T}_\sigma \to \mathcal{T}_A^{\oslash}$

- $(!^i\alpha)^\bullet = (i + \alpha') \oslash \alpha$, where $i \geq 0$, $\alpha' \in \mathcal{V}_{vn}$ is fresh,
- $(!^i(A \odot B))^\bullet = (i+\alpha) \oslash ((A)^\bullet \odot (B)^\bullet)$, where $i \geq 0$, $\alpha \in \mathcal{V}_{vn}$ is fresh and $\odot \in \{\multimap, \otimes, \boxtimes\}$,
- $(!^i H)^\bullet = (i+\alpha) \oslash H$ where $i \geq 0$, $\alpha \in \mathcal{V}_{vn}$ is fresh and $H \in \{\mathbb{S}_i^{!j} \mid, i, j \in \mathbb{N}\} \cup \{\mathbb{N}^{!j} \mid j \in \mathbb{N}\}$.

The transformation back $[\![\cdot]\!] : \mathcal{T}_A^{\oslash} \to \mathcal{T}_\sigma$ is defined as

- $[\![\alpha \oslash \alpha']\!] = \alpha'$,
- $[\![(n) \oslash B]\!] = !^n [\![B]\!]$, where $!^0 A = A$,
- $[\![\alpha \oslash (A \odot B)]\!] = [\![A]\!] \odot [\![B]\!]$, where $\odot \in \{\multimap, \otimes, \boxtimes\}$,
- $[\![\alpha \oslash H]\!] = H$ where $\alpha \in \mathcal{V}_{vn}$ and $H \in \{\mathbb{S}_i^{!j} \mid, i, j \in \mathbb{N}\} \cup \{\mathbb{N}^{!j} \mid j \in \mathbb{N}\}$.

Note that the translation back is correct only in case the translation in the context $[\![\alpha \oslash (A \multimap N \oslash C)]\!]$ is always applied so that $N = 0$. A substitution S is *proper wrt. a set of expressions* E when $S(N) = 0$ in the subexpressions of the form $\alpha \oslash (A \multimap N \oslash C)$ of expressions in E. The operations $(\cdot)^\bullet$ and $[\![\cdot]\!]$ extend to environments so that $(\Gamma)^* = \{x : (A)^* \mid x : A \in \Gamma\}$ where $(\cdot)^* \in \{(\cdot)^\bullet, [\![\cdot]\!]\}$.

The intuition behind the expressions presented here is that they make possible to more explicitly control the ! modalities. These must be, however, controlled in a non-standard way which cannot be handled with first-order unification techniques. The unification has the usual first-order ingredient, but to control the numbers of ! in $(\multimap E)$ and (let) rules we need a sort of second-order operation that can handle the presence of (sp) rules to obtain the type of the argument (see the definition of E_4 in the rule of $(\multimap E)$ in Fig. 5). The number cannot be handled locally since an occurrence of a variable in a different part of a derivation may require a higher number of (sp) that is immediately visible in the currently handled rule. Therefore, we split unification into two parts, i.e. one tractable by first-order techniques and one that operates on numerals and must be solved globally after the global information on the use of the (sp) rule is gathered. This separation requires a more subtle definition of the most general unifier operation. This is presented in the definition below.

▶ **Definition 13** (most general unification pair)**.** The operation $\mathsf{mgu}(\cdot) : \mathcal{E} \times \mathsf{Subst} \to \mathsf{Subst} \times \mathcal{E}_{\mathbb{N}} \cup \{\mathsf{fail}\}$, where $\mathcal{E}$ is the set of sets of pairs $A \doteq A'$ with $A, A' \in \mathcal{T}_A^{\oslash} \cup \mathcal{T}_B^{\oslash}$, Subst is the set of substitutions $\mathcal{V}_{vl} \rightharpoonup \mathcal{T}_C^{\oslash}$, and $\mathcal{E}_{\mathbb{N}}$ is the set of sets of pairs $B \doteq B'$ with $B, B' \in \mathcal{T}_B^{\oslash}$, is defined inductively as follows:

- $\mathsf{mgu}(\{A_1 \odot A_1' \doteq A_2 \boxdot A_2'\} \cup E, U_0) = \mathsf{fail}$ when $\odot \neq \boxdot$,
- $\mathsf{mgu}(\{N_1 \oslash A_1 \doteq N_2 \oslash A_2\} \cup E, U_0) = \mathsf{fail}$ when $\mathsf{mgu}(\{A_1 \doteq A_2\} \cup E, U_0) = \mathsf{fail}$,
- $\mathsf{mgu}(\{N_1 \oslash A_1 \doteq N_2 \oslash A_2\} \cup E, U_0) = (U, E' \cup \{N_1 \doteq N_2\})$ when $\mathsf{mgu}(\{A_1 \doteq A_2\} \cup E, U_0) = (U, E')$,
- $\mathsf{mgu}(\{A_1 \multimap N_1 \oslash A_1' \doteq A_2 \multimap N_2 \oslash A_2'\} \cup E, U_0) = \mathsf{fail}$ when $\mathsf{mgu}(\{A_1 \doteq A_2, A_1' \doteq A_2'\} \cup E, U_0) = \mathsf{fail}$,
- $\mathsf{mgu}(\{A_1 \multimap N_1 \oslash A_1' \doteq A_2 \multimap N_2 \oslash A_2'\} \cup E, U_0) = (U, E' \cup \{N_1 \doteq 0, N_2 \doteq 0\})$ when $\mathsf{mgu}(\{A_1 \doteq A_2, A_1' \doteq A_2'\} \cup E, U_0) = (U, E')$,
- $\mathsf{mgu}(\{N_1 \oslash A_1 \odot N_1' \oslash A_1' \doteq N_2 \oslash A_2 \odot N_2' \oslash A_2'\} \cup E, U_0) = \mathsf{fail}$ when $\mathsf{mgu}(\{A_1 \doteq A_2, A_1' \doteq A_2'\} \cup E, U_0) = \mathsf{fail}$ for $\odot \in \{\otimes, \boxtimes\}$,
- $\mathsf{mgu}(\{N_1 \oslash A_1 \odot N_1' \oslash A_1' \doteq N_2 \oslash A_2 \odot N_2' \oslash A_2'\} \cup E, U_0) = (U, E' \cup \{N_1 \doteq 0, N_2 \doteq 0, N_1' \doteq 0, N_2' \doteq 0\})$ when $\mathsf{mgu}(\{A_1 \doteq A_2, A_1' \doteq A_2'\} \cup E, U_0) = (U, E')$ for $\odot \in \{\otimes, \boxtimes\}$,
- $\mathsf{mgu}(\{C_1 \doteq C_2\} \cup E, U_0) = \mathsf{fail}$ when C_1, C_2 are different type constants
- $\mathsf{mgu}(\{C \doteq C\} \cup E, U_0) = \mathsf{mgu}(E, U_0)$ when C is a type constant,

- $\mathsf{mgu}(\{\alpha \doteq A\} \cup E, U_0) = \mathsf{fail}$ when $A \neq \alpha$ and α occurs in A,
- $\mathsf{mgu}(\{\alpha \doteq A\} \cup E, U_0) = \mathsf{mgu}(E[A/\alpha], [A/\alpha] \circ U_0)$ when $A = \alpha$ or α does not occur in A,
- $\mathsf{mgu}(\emptyset, U_0) = (U_0, \emptyset)$.

By default $\mathsf{mgu}(E) = (U, E)$ where $\mathsf{mgu}(E, \emptyset) = (U', E)$ and $U = R \circ U'$ where R is a renaming of all variables in $\mathrm{dom}(U)$ to fresh variables.

This most general unification pair enjoys the following natural property:

▶ Proposition 14 (correctness and completness of $\mathsf{mgu}(\cdot)$).

- If $\mathsf{mgu}(E) = (U, E')$ and E' is solvable with U' proper wrt. E then for each $A \doteq A' \in E$ where $A, A' \in \mathcal{T}_A^{\oslash} \cup \mathcal{T}_B^{\oslash}$ it holds that $U'(U(A)) = U'(U(A'))$.
- If there is U proper wrt. E such that $Z(U(A)) = Z(U(A'))$ for each $A \doteq A' \in E$ where $A, A' \in \mathcal{T}_A^{\oslash} \cup \mathcal{T}_B^{\oslash}$ and $Z(\alpha) = 0$ for each $\alpha \in \mathcal{V}_{vn}$ then $\mathsf{mgu}(E) = (U_1, E_1)$ and there is a substitution $U' : \mathcal{V}_{vl} \rightharpoonup \mathcal{T}_A^{\oslash}$ and a solution U'' of E_1 such that for each $\alpha \in \mathrm{dom}(U)$ the equality $Z(U(\alpha)) = Z(U'(U_1(U''(\alpha))))$ holds.

Proof. A standard proof is left to the reader. ◀

The main technical lemma that describes the operation of W-lin looks as follows.

▶ **Lemma 15** (key lemma).

1. *If* $\Gamma \vdash M : A \rightsquigarrow (U_1, E_1)$ *then* $\mathrm{FTV}(\Gamma, A) \cap \mathrm{FTV}(\{U_1(\alpha) | \alpha \in \mathrm{dom}(U_1)\}) = \emptyset$.
2. *For any term M and context Γ at most one rule in Fig. 5 can be used.*
3. *For any term M, context Γ, and type σ if a rule in Fig. 5 is applied then for each of the premises $\Gamma' \vdash M' : \sigma'$ and $x : \tau \in \Gamma'$ we have $\tau = B \oslash C$.*
4. *If W-lin started with $(\Gamma)^\bullet \vdash M : (A)^\bullet$ returns $(\Gamma)^\bullet \vdash M : (A)^\bullet \rightsquigarrow (U_1, E_1)$, and E_1 is unifiable with $U_1^{\#}$ then for $U = U_1^{\#} \circ U_1$ it holds that $[\![U(\Gamma)]\!] \vdash_{\mathrm{MLSTA}} M : [\![U(A)]\!]$. Moreover, the number of rules in the run of W-lin is the same as the number of rules different than (sp) in the resulting derivation in MLSTA.*
5. *Let Γ be a context and M a term. If there is a substitution U such that $U(\Gamma) \vdash_{\mathrm{MLSTA}} M : U(A)$ then the algorithm W-lin infers $(\Gamma)^\bullet \vdash M : (A)^\bullet \rightsquigarrow (U_1, E_1)$, the set E_1 is unifiable by $U_1^{\#}$, and there is a substitution U' such that for each variable $\beta \in \mathrm{dom}(U)$ $Z((U(\beta))^\bullet) = Z(U'(U_1^{\#}(U_1(\beta))))$. Moreover, the number of rules other than (sp) in a derivation in MLSTA is the same as the number of rules different than (sp) in the resulting run of W-lin.*
6. *For each Γ, M, A the algorithm W-lin terminates.*

▶ **Theorem 16** (decidability of TCP, TIP, and TP). *The TCP, TIP, and TP for the system MLSTA are decidable.*

Proof. We use here the algorithm W-lin. Note that is always terminating by Lemma 15(6).

In case of TCP we are given a context Γ, a term M, and a type σ. We may assume that σ does not start with ! by Corollary 3. Then we apply W-lin with the input $(\Gamma)^\bullet \vdash M : (\sigma)^\bullet$. In case this is derivable we obtain by Lemma 15(5) a pair (U, E) where E is unifiable with some $U^{\#}$. We now see that $Z(\emptyset(U^{\#}(U((\Gamma)^\bullet))) \vdash M : Z(\emptyset(U^{\#}(U((\sigma)^\bullet))))$ is derivable in MLSTA, but this is exactly the initial judgement as there are no variables in Γ, σ. In case this is not derivable by W-lin the initial judgement cannot be derivable in MLSTA by Lemma 15(4).

In case of TIP and TP we proceed in the same way, but we introduce substitutable variables for types the existence of which we have to discover, namely for the resulting type in case of TIP, and for the resulting type and the types in the context in case of TP. In case of TP we have to, in addition, guess which variables in the context should have type schemes. These variables must be packed by suitable `let` expression, essentially to manage the polymorphism in the way compatible with W-lin. ◀

7 Conclusions and Further Work

The system MLSTA we propose here can be viewed, similarly as ML in relation to System F, as a kind of interface over the system with full polymorphism, STA. The system offers a reasonable polymorphism with algebraic data structures such as naturals, booleans, and strings as well as recursion over the data types. All these features have their impredicative counterparts in STA. This view suggests a number of enhancements that can be done. One could develop the full theory of algebraic data types in our MLSTA, in particular polymorphic lists or polymorphic binary trees. Another possible improvement is to introduce more flexibility in the use of available constants. Currently, the programmer must provide the numerical parameters such as j_1, j_2 in `add` that express the level of natural numbers the addition operates in. One can extend our rule (AxC) to include the automatic calculation of the indexes. At last one can try to exploit other systems such as STA_+ or STA_B [12] and give their ML-like versions.

References

1. Andrea Asperti and Luca Roversi. Intuitionistic light affine logic. *ACM Trans. Comput. Logic*, 3:137–175, January 2002.
2. Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. A polytime functional language from light linear logic. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010*, volume 6012 of *LNCS*, pages 104–124. Springer-Verlag, 2010.
3. Patrick Baillot and Martin Hofmann. Type inference in intuitionistic linear logic. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 219–230. ACM, 2010.
4. Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus. *Information and Computation*, 207:41–62, January 2009.
5. Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, December 1992.
6. Jacek Chrząszcz and Aleksy Schubert. The role of polymorphism in the characterisation of complexity by soft types. In Piotr Sankowski and Filip Murlak, editors, *Mathematical Foundations of Computer Science — 36th International Symposium, MFCS 2011*, volume 6907 of *LNCS*, pages 219–230. Springer-Verlag, 2011.
7. Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North-Holland, 1964.
8. Paolo Coppola, Ugo Dal Lago, and Simona Ronchi Della Rocca. Light logics and the call-by-value lambda calculus. *Logical Methods in Computer Science*, 4(4), 2008.
9. Ronald Fagin. *Contriburions to the model theory of'finire srructures.* PhD thesis, University of California at Berkeley, 1973.
10. Marco Gaboardi and Simona Ronchi della Rocca. Type inference for a polynomial lambda calculus. In Stefano Berardi, Ferruccio Damiani, and Ugo De'Liguoro, editors, *Types for Proofs and Programs, International Conference, TYPES 2008*, volume 5497 of *LNCS*, pages 136–152. Springer-Verlag, 2009.
11. Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. A logical account of PSPACE. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'08, pages 121–131. ACM, 2008.

12 Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. Soft linear logic and polynomial complexity classes. *ENTCS*, 205:67–87, April 2008.

13 Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. An implicit characterization of pspace. *ACM Trans. Comput. Logic*, 13(2):18:1–18:36, April 2012.

14 Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for λ-calculus. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL*, volume 4646 of *LNCS*, pages 253–267. Springer-Verlag, 2007.

15 Gemalto. *Java Card™ & STK Applet Development Guidelines*. Gemalto, 2009.

16 Jean-Yves Girard. Light linear logic. *Information and Computation*, 143:175–204, June 1998.

17 Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Trans. Program. Lang. Syst.*, 15(2):211–252, April 1993.

18 Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, August 1987.

19 Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318:163–180, June 2004.

20 Ugo Dal Lago and Patrick Baillot. On light logics, uniform encodings and polynomial time. *Mathematical Structures in Comp. Sci.*, 16(4):713–733, August 2006.

21 Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010*, volume 6012 of *LNCS*, pages 205–225. Springer-Verlag, 2010.

22 Ugo Dal Lago and Ulrich Schöpp. Type inference for sublinear space functional programming. In Kazunori Ueda, editor, *Programming Languages and Systems — 8th Asian Symposium, APLAS 2010*, volume 6461 of *LNCS*, pages 376–391. Springer-Verlag, 2010.

23 Daniel Leivant. Stratified functional programs and computational complexity. In Mary S. Van Deusen and Bernard Lang, editors, *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 325–333. ACM, 1993.

24 Harry Mairson and Kazushige Terui. On the computational complexity of cut-elimination in linear logic. In Carlo Blundo and Cosimo Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS 2003*, volume 2841 of *LNCS*, pages 23–36. Springer-Verlag, 2003.

25 François Maurel. Nondeterministic light logics and NP-time. In Martin Hofmann, editor, *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003*, volume 2701 of *LNCS*, pages 241–255. Springer-Verlag, 2003.

26 Wojciech Mostowski. Rigorous development of JavaCard applications. In T. Clark, A. Evans, and K. Lano, editors, *Proceedings of Fourth Workshop on Rigorous Object-Oriented Methods*, London, 2002.

27 Christos H. Papadimitriou. A note on the expressive power of Prolog. *Bulletin of the EATCS*, pages 21–22, 1985.

28 J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

29 David A. Schmidt. *The Structure of Typed Programming Languages*. The MIT Press, 1994.

30 Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science, LICS 2007*, pages 411–420. IEEE Computer Society, 2007.