Synthesis of Functional Programs with Help of First-Order Intuitionistic Logic*

Marcin Benke[1], Aleksy Schubert[2], and Daria Walukiewicz-Chrząszcz[3]

1 **Institute of Informatics, University of Warsaw, Warsaw, Poland**
ben@mimuw.edu.pl
2 **Institute of Informatics, University of Warsaw, Warsaw, Poland**
alx@mimuw.edu.pl
3 **Institute of Informatics, University of Warsaw, Warsaw, Poland**
daria@mimuw.edu.pl

Abstract

Curry-Howard isomorphism makes it possible to obtain functional programs from proofs in logic. We analyse the problem of program synthesis for ML programs with algebraic types and relate it to the proof search problems in appropriate logics. The problem of synthesis for closed programs is easily equivalent to the proof construction in intuitionistic propositional logic and thus fits in the class of PSPACE-complete problems. We focus further attention on the synthesis problem relative to a given external library of functions. It turns out that the problem is undecidable for unbounded instantiation in ML. However its restriction to instantiations with atomic types only results in a case equivalent to proof search in a restricted fragment of intuitionistic first-order logic, being the core of Σ_1 level of the logic in the Mints hierarchy. This results in EXPSPACE-completeness for this special case of the ML program synthesis problem.

1998 ACM Subject Classification D.1.2 Automatic Programming, F.4.1 Mathematical Logic, I.2.2 Automatic Programming

Keywords and phrases ML, program synthesis

Digital Object Identifier 10.4230/LIPIcs.FSCD.2016.12

1 Introduction

In general, program synthesis is the problem of the following form: given a not necessarily executable specification, find an executable program satisfying that specification. The idea of mechanically constructed programs or more precisely programs correct-by-construction appeared already a long time ago and not only in functional programming but also in imperative programming [22, 5] and in logic programming. This idea arises naturally in the context of increasing demand for programmer's productivity.

In 2005 Augustsson created Djinn [1], "a small program that takes a (Haskell) type and gives you back a function of that type if one exists." As an example supporting usefulness of such program extraction, he used the key functions of the continuation monad:

```
# return, bind, and callCC in the continuation monad
   Djinn> type C a = (a -> r) -> r
   Djinn> returnC ? a -> C a
```

* This work is supported by NCN grant DEC-2012/07/B/ST6/01532.

Given this query he obtained an answer:

```
returnC :: a -> C a
returnC x1 x2 = x2 x1
```

Moreover, for

```
Djinn> bindC ? C a -> (a -> C b) -> C b
```

he obtained

```
bindC :: C a -> (a -> C b) -> C b
bindC x1 x2 x3 = x1 (\ c15 -> x2 c15 (\ c17 -> x3 c17))
```

and finally for

```
Djinn> callCC ? ((a -> C b) -> C a) -> C a
```

he got

```
callCC :: ((a -> C b) -> C a) -> C a
callCC x1 x2 = x1 (\ c15 _ -> x2 c15) (\ c11 -> x2 c11)
```

Indeed, in certain situations such as the one above, there exists only a handful of functions of a given type (sometimes — barring usage of functions such as *error* or *undefined* — just one). If the type is complex and specific enough, we may be content with any of them. In such cases the programmer should be liberated from the effort of coding and the code of the program should be given for acceptance as soon as the type of the function is given.

The reference point for type systems in functional programming languages with static typechecking is the model language of ML [9, 4]. This language brings the `let` $x = M$ `in` N construct into the inventory of program term constructs available in the standard Curry-style λ-calculus. This makes it possible to assign to the term M a polymorphic type scheme $\forall\alpha_1 \ldots \alpha_n.\tau$ and use it within N in various places so that in each of them $\alpha_1, \ldots, \alpha_n$ can be instantiated with different concrete types. This design of language leads to the situation that formal typing judgements use contexts that in addition to assertions $x : A$ about (monomorphic) types contain assertions about type schemes $x : \tau$.

However, the expressivity of ML types is very limited. Stronger type theories were applied to extend the reasoning possibilities for functional programs, including calculus of constructions [3] or Martin-Löf type theory [12]. Types correspond there to propositions in richer logics and one can, for example, specify sorting as

$$\forall x\ \exists y\ \text{ordered}(y) \wedge \text{permutation}(x, y)$$

A constructive proof of this specification should be turned into a sorting procedure by a program synthesis mechanisms. This view is supported by the Curry-Howard isomorphism, which identifies a proof (of a specification) with a program (meeting that specification).

In richer type systems the relation between proofs and programs is not simple and can take up the form of a program extraction procedure. In its course, all "computationally irrelevant" content is deleted while the remaining executable parts of the proof are guaranteed to be correct with respect to the specification formula proved by the initial proof.

Program extraction procedures are typically associated with proof assistants. In particular, Minlog [18], Isabelle/HOL [14], Coq [2] have such mechanisms. Let us examine closer their features based upon the extraction mechanism of Coq, designed by Paulin [15] and Letouzey [10]. The input for the extraction mechanism are Coq proofs and functions. The extracted

programs are expressed in functional languages such as Ocaml, Haskell and Scheme. One delicate point concerns the typability of the extracted code, since neither Haskell nor Ocaml have dependent types. A first solution is to use a type-free language like Scheme. Another possibility is to use ML-like typing as long as possible and insert some unsafe type coercions when needed: `Obj.magic` in Ocaml and `unsafeCoerce` in Haskell. This feature implies that resulting types of functions may go beyond the realm of tautologies of the base logic associated with the ML type system.

In this paper we are interested in the problem of program generation for functional programs. For this we focus on the model language of ML. In its basic form, the problem has already been fully exploited in Djinn. However, the example of program generation with help of Djinn above used no external context of library functions, i.e. no additional symbols were available that could occur in the generated program except from those explicitly declared in the body of the generated function. A more realistic case is when the programmer expects that the program to be created should contain certain symbols that were defined beforehand in the available program libraries. This leads to the *problem of synthesis* for ML:

> Given a set Γ of library functions together with their types and a type τ of the goal program, find a term M that has type τ under the context Γ.

In the current paper we analyse unrestricted problem of program synthesis for ML with algebraic types. The problem turns out to be undecidable when we allow Γ to contain not only constructive tautologies, but symbols of arbitrary type. We further consider ML with restricted type instantiations so that types can be instantiated only with atomic types. We can prove that this case is equivalent to proof search for a restricted fragment of intuitionistic first-order logic, being the core of Σ_1 level of the logic in the Mints hierarchy. As a result we obtain EXPSPACE-completeness for the ML program synthesis problem for so constrained instantiations.

The current paper is constructed as follows. In Section 2 the type systems and logics used in the paper are defined. The problem of program synthesis is studied in Section 3. In Subsection 3.1 we present the undecidability proof for ML with unrestricted instantiations and in Subsection 3.2 we analyse the situation when the instantiations are restricted to atomic types. We present conclusions together with possible further work directions in Section 4.

2 Presentation of Logical Systems

2.1 The System of ML

We assume that an infinite set of object variables *Vars* is available. The expressions of ML are defined with the following grammar

$$M ::= x \mid \lambda x.M \mid M_1 M_2 \mid \texttt{let}\ x = M_1\ \texttt{in}\ M_2$$

where $x \in \mathit{Vars}$. The set $\mathrm{FV}(M)$ of *free variables in a term* M is defined inductively as

- $\mathrm{FV}(x) = \{x\}$, $\mathrm{FV}(\lambda x.M) = \mathrm{FV}(M)\backslash\{x\}$, $\mathrm{FV}(M_1 M_2) = \mathrm{FV}(M_1) \cup \mathrm{FV}(M_2)$, $\mathrm{FV}(\texttt{let}\ x = M_1\ \texttt{in}\ M_2) = \mathrm{FV}(M_1) \cup (\mathrm{FV}(M_2)\backslash\{x\})$.

This notion is extended naturally to sets of terms. The capture avoiding substitution that assigns M to a variable x is written $[x := M]$ and as usual extended to $[x_1 := M_1, \ldots, x_n := M_n]$ when many variables are involved. For the definition of types we assume that we have a finite, but of unbounded size, set of type constants *TConst* as well as an infinite set *TVars* of type variables. The types and type schemes are defined with the grammar

$$A ::= o \mid \alpha \mid A \to A \quad \text{(types)} \qquad \sigma ::= A \mid \forall\alpha.\sigma \quad \text{(type schemes)}$$

$$\frac{A \text{ inst } \sigma}{\Gamma, x : \sigma \vdash x : A}\ (var)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ (\to I) \qquad \frac{\Gamma \vdash M_1 : A \to B \quad \Gamma \vdash M_2 : A}{\Gamma \vdash M_1 M_2 : B}\ (\to E)$$

$$\frac{\Gamma \vdash M_1 : B \quad \Gamma, x : \mathsf{gen}(\Gamma, B) \vdash M_2 : A}{\Gamma \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2 : A}\ (\texttt{let})$$

Figure 1 Rules of ML.

where $o \in \mathit{TConst}, \alpha \in \mathit{TVars}$. A *context* Γ is a finite set of pairs $x : \sigma$ such that when $x : \sigma$ and $y : \tau$ occur in Γ then $x \neq y$. The set $\mathrm{FTV}(\sigma)$ of *free type variables in type* σ is defined inductively as

- $\mathrm{FTV}(o) = \emptyset$, $\mathrm{FTV}(\alpha) = \{\alpha\}$, $\mathrm{FTV}(A \to B) = \mathrm{FTV}(A) \cup \mathrm{FTV}(B)$, $\mathrm{FTV}(\forall\alpha.\sigma) = \mathrm{FTV}(\sigma) \backslash \{\alpha\}$.

This notion extends naturally to sets of types and to contexts.

The relation $A \text{ inst } \forall\alpha_1 \ldots \alpha_n.A'$ holds when there is a (capture avoiding) substitution S that assigns types to type variables with $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{dom}(S)$, where $\mathrm{dom}(S)$ is the domain of S, such that $A = S(A')$. The function $\mathsf{gen}(\cdot)$ is defined as $\mathsf{gen}(\Gamma, B) = \forall\alpha_1 \ldots \alpha_n.B$ where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(B) \backslash \mathrm{FTV}(\Gamma)$. The type assignment rules for the system are presented in Figure 1. We write $\Gamma \vdash_{\mathrm{ML}} M : \tau$ to tell that the judgement $\Gamma \vdash M : \tau$ is derivable according to these rules.

This language can be extended to handle *algebraic types*. In that case we assume that there is a finite set of algebraic type constructors *TAlg*, and each $P \in \mathit{TAlg}$ has arity given by $\mathit{arity}(P)$. The types are formed according to the following grammar

$$\begin{array}{ll} A ::= o \mid \alpha \mid A \to A \mid P(A_1, \ldots, A_n) & \text{(types)} \\ \sigma ::= A \mid \forall\alpha.\sigma & \text{(type schemes)} \end{array}$$

where $o \in \mathit{TConst}, \alpha \in \mathit{TVars}$, $P \in \mathit{TAlg}$ and $\mathit{arity}(P) = n$. The algebraic types come usually equipped with term constants that make it possible to construct values of the algebraic types and destruct them. We omit the constructors from our account since they can be introduced to our setting as polymorphic object variables placed in contexts (however, the presence of them may change the complexity bounds, as the presence of any other class of variables can do). The algebraic type of the form $P(A_1, \ldots, A_n)$, type constant or type variable are called *atomic types*. If any of them occurs at the end of a type or type scheme, it is called the *target* of the type or type scheme, respectively.

This system is accompanied by a reduction relation $\to_\beta$ that is defined by the following redexes

$$(\lambda x.M)N \quad \to_\beta \quad M[x := N], \qquad\qquad \texttt{let}\ x = M\ \texttt{in}\ N \quad \to_\beta \quad N[x := M]$$

extended by syntactic closure. The transitive-reflexive closure of $\to_\beta$ is $\to_\beta^*$. The *normal form* of a term M is defined as a term $\mathrm{NF}(M)$ such that there is no M' such that $\mathrm{NF}(M) \to_\beta M'$. The system of ML enjoys the following proof-theoretic properties:

▶ **Theorem 1.**

- **(Subject reduction)** *If* $\Gamma \vdash_{\mathrm{ML}} M : A$ *and* $M \to_\beta N$ *then* $\Gamma \vdash_{\mathrm{ML}} N : A$.
- **(Church-Rosser)** *If* $M \to_\beta^* N_1$ *and* $M \to_\beta^* N_2$ *then there is a term* M' *such that* $N_1 \to_\beta^* M'$ *and* $N_2 \to_\beta^* M'$.
- **(Strong normalisation)** *For each term* N *such that* $\Gamma \vdash_{\mathrm{ML}} N : A$ *there is no infinite sequence* M_i *for* $i \in \mathbb{N}$ *such that* $N = M_0$ *and* $M_i \to_\beta M_{i+1}$ *for* $i \in \mathbb{N}$.

$$\frac{A \ \mathsf{inst}_1 \ \sigma}{\Gamma, x : \sigma \vdash x : A} \ (var)$$

Figure 2 The modified rule (var) of ML_1.

$$\frac{}{\Gamma, x : A \vdash x : A} \ (var)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \ (\to I) \qquad \frac{\Gamma \vdash M_1 : A \to B \quad \Gamma \vdash M_2 : A}{\Gamma \vdash M_1 M_2 : B} \ (\to E)$$

Figure 3 Rules of the simply-typed lambda calculus.

Proof. The subject reduction property can be attributed to Dubois [6]. The rest is obtained as a folklore result resulting from an obvious embedding of the system to System F of Girard and Reynolds [8, 16]. ◀

By a straightforward inspection of cases we obtain the following proposition that describes the set of normal forms in ML.

▶ **Proposition 2.**

1. *If M is an* ML *term in normal form then $M = x$, $M = xM_1 \dots M_n$ for some $n \geq 1$, or $M = \lambda x.M_1$ where $M_1, \dots, M_n$ are in normal form.*
2. *If $\Gamma \vdash_{\mathrm{ML}} M : A \to B$ is in normal form then*
   a. *either $M = xM_1 \cdots M_n$ for $n \geq 0$ with $x : \forall\alpha_1 \dots \alpha_k.A_1 \to \cdots \to A_n \to C \in \Gamma$, $A \to B = C[\alpha_1 := B_1, \dots, \alpha_k := B_k]$ for some $B_1, \dots, B_k$ and $\Gamma \vdash M_i : A_i[\alpha_1 := B_1, \dots, \alpha_k := B_k]$ for $i = 1, \dots, n$,*
   b. *or $M = \lambda x.M_0$ where $\Gamma, x : A \vdash_{\mathrm{ML}} M_0 : B$.*
3. *If $\Gamma \vdash_{\mathrm{ML}} M : A$ where A is atomic type and M is in normal form then $M = xM_1 \cdots M_n$ for $n \geq 0$ with $x : \forall\alpha_1 \dots \alpha_k.A_1 \to \cdots \to A_n \to C \in \Gamma$, $A = C[\alpha_1 := B_1, \dots, \alpha_k := B_k]$ for some $B_1, \dots, B_k$ and $\Gamma \vdash M_i : A_i[\alpha_1 := B_1, \dots, \alpha_k := B_k]$ for $i = 1, \dots, n$.*

Proof. Standard arguments are left to the reader. ◀

Observe that normal forms in this system have no occurrences of the `let` $\cdot$ `in` $\cdot$ construct.

We consider here in more detail a restricted version of the system ML, namely ML_1, in which the terms, types and type schemes remain the same as in ML, but the instantiation relation inst is restricted to inst_1 where $A \ \mathsf{inst}_1 \ \forall\alpha_1 \dots \alpha_n.A'$ holds whenever there is a substitution S such that $\{\alpha_1, \dots, \alpha_n\} = \mathrm{dom}(S)$, $A = S(A')$, and for each $\alpha \in \mathrm{dom}(S)$ we have $|S(\alpha)| = 1$. In this system only one rule is modified, namely the (var) rule, and it takes the form presented in Figure 2.

Simply-typed Lambda Calculus. Simply-typed lambda calculus may be viewed as a restriction of ML, the terms and types of which respectively are

$$M ::= x \mid \lambda x.M \mid M_1 M_2 \qquad A ::= o \mid \alpha \mid A \to A$$

The rules are presented in Figure 3. Note that the (var) rule is a special case of the ML (var) rule for the empty string of quantifiers (i.e. when the instantiated type scheme is a type). Also note that the rule for `let` $\cdot$ `in` $\cdot$ is missing. We write $\Gamma \vdash_{\to} M : \tau$ to tell that the judgement $\Gamma \vdash M : \tau$ is derivable according to these rules in Figure 3.

$$\frac{}{\Gamma, x : \varphi \ \vdash x : \varphi}\ (Ax)$$

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash \lambda x : \varphi.M \ : \ \varphi \to \psi}\ (\to I) \qquad \frac{\Gamma \vdash M : \varphi \to \psi \quad \Gamma \vdash N : \varphi}{\Gamma \vdash MN : \psi}\ (\to E)$$

$$\frac{\Gamma \vdash M : \varphi}{\Gamma \vdash \lambda X\, M : \forall X \varphi}\ (\forall I)^* \qquad \frac{\Gamma \vdash M : \forall X \varphi}{\Gamma \vdash MY : \varphi[X := Y]}\ (\forall E)$$

* This rule is subject to the standard eigenvariable condition.

Figure 4 Proof assignment rules for intuitionistic first-order logic with $\to, \forall$.

2.2 Intuitionistic First-order Logic

For the definition of formulas in intuitionistic first-order logic (IFOL) we need an infinite set of object variables $Vars_{IFOL}$ as well as a finite, but of unbounded size set of predicates $Preds$ such that each $P \in Preds$ has arity given by $arity(P)$ (we abuse the notation and use the same metavariables for predicates and algebraic types as they are translated bijectively in our approach, this also explains the overloading of the function $arity(\cdot)$). Formulas of intuitionistic first-order logic with $\to, \forall$ are built using the following grammar

$$\varphi ::= P(X_1, \ldots, X_n) \mid \varphi_1 \to \varphi_2 \mid \forall X\, \varphi$$

where $X, X_1, \ldots, X_n \in Vars_{IFOL}$. In this article we use formulas of restricted form that can be described within the framework of the Mints hierarchy [17] where the presence of a formula on a particular level depends on the form of its classical prenex form. We can define syntactically the resulting classes of formulas $\mathbf{\Sigma}_n, \mathbf{\Pi}_n$ in the following way. The sets $\mathbf{\Sigma}_0 = \mathbf{\Pi}_0$ are equal to the set of quantifier-free formulas. Further,

$$\mathbf{\Sigma}_{n+1} ::= P(X_1, \ldots, X_n) \mid \mathbf{\Pi}_n \mid \mathbf{\Pi}_{n+1} \to \mathbf{\Sigma}_{n+1}$$
$$\mathbf{\Pi}_{n+1} ::= P(X_1, \ldots, X_n) \mid \mathbf{\Sigma}_n \mid \mathbf{\Sigma}_{n+1} \to \mathbf{\Pi}_{n+1} \mid \forall X\, \mathbf{\Pi}_{n+1}$$

where $X_1, \ldots, X_n, X \in Vars_{IFOL}$. We introduce an additional class of formulas in natural form defined by the following grammar from the symbol $\mathbf{N}$

$$\mathbf{N} ::= \mathbf{O} \mid \mathbf{B} \to \mathbf{N} \qquad \mathbf{B} ::= \mathbf{O} \mid \forall X\, \mathbf{B} \qquad \mathbf{O} ::= P(X_1, \ldots, X_n) \mid \mathbf{O} \to \mathbf{O}$$

where $X_1, \ldots, X_n, X \in Vars_{IFOL}$. The formulas are natural in the sense that they avoid nested quantifiers, and people tend to avoid internal quantification. Again the predicate $P(X_1, \ldots, X_n)$, where $n \geq 0$, at the end of a formula is called *target* of the formula. We observe that $\mathbf{N} \subseteq \mathbf{\Sigma}_1$.

The proofs for valid formulas of the logic can be represented by terms. To define them we need an infinite set of proof variables $Vars_P$. The terms are generated by the grammar

$$M ::= x \mid \lambda x : \varphi.M \mid M_1 M_2 \mid \lambda X\, M \mid MX$$

where $x \in Vars_P$ and $X \in Vars_{IFOL}$. The proof assignment rules for the logic are presented in Figure 4.

This system is again accompanied by a reduction relation $\to_\beta$ that is defined through the redex $(\lambda x.M)N \to_\beta M[x := N]$ as well as $(\lambda X M)Y \to_\beta M[X := Y]$ extended by syntactic closure. The transitive-reflexive closure of $\to_\beta$ is $\to_\beta^*$. The *normal form* of a term M is defined as a term $\mathrm{NF}(M)$ such that there is no M' with $\mathrm{NF}(M) \to_\beta M'$. We also define the notion of a proof term in *long normal form*, abbreviated *lnf*.

- If N is an lnf of type φ then $\lambda X\, N$ is an lnf of type $\forall X\, \varphi$.
- If N is an lnf of type ψ then $\lambda x : \varphi.\, N$ is an lnf of type $\varphi \to \psi$.
- If $N_1, \ldots, N_n$ are lnf or object variables, and $xN_1 \ldots N_n$ is of an atom type, then $xN_1 \ldots N_n$ is an lnf.

We have now (see e.g. the paper by Schubert et al [17]) the following basic properties.

▶ **Proposition 3.**

1. *If φ is intuitionistically derivable from Γ then $\Gamma \vdash N : \varphi$, for some lnf N.*
2. *If $\Gamma \vdash N : P(\vec{x})$, where $P(\vec{x})$ is an atomic formula and N is an lnf, then $N = X\vec{D}$, where $(X : \psi) \in \Gamma$ with $target(\psi) = P$, and $\vec{D}$ is a sequence that may contain proof terms and object variables.*

▶ **Problem 4** (Provability Problem)**.** *Given a context Γ and formula φ check if there is a proof term M such that $\Gamma \vdash_{\mathrm{IFOL}} M : \varphi$ holds.*

This problem is known to be EXPSPACE-complete for formulas in $\mathbf{\Sigma}_1$. We have even more.

▶ **Theorem 5.** *The provability problem when formulas are restricted to come from $\mathbf{\Sigma}_1$ is EXPSPACE-complete. The same holds for formulas in $\mathbf{\Sigma}_1 \cap \mathbf{N} = \mathbf{N}$.*

The paper by Schubert et al [17] states the above result only for $\mathbf{\Sigma}_1$. However, the hardness proof uses formulas from $\mathbf{N}$ so the result holds for this restricted class.

3 The Problem of Synthesis

The program synthesis problem in its most basic formulation is as follows

▶ **Problem 6** (Closed Program Synthesis for ML)**.** *Given a type A of* ML *check if there is a term M such that $\vdash_{\mathrm{ML}} M : A$.*

We can restate it in the vocabulary of programming languages as follows: given a type τ find a program M that has the type.

One may be tempted to ask why we demand type instead of type scheme in the problem, but this is easily explained by the fact that there are no terms of any type scheme in the language of ML. Functional programming languages make it possible to define functions for type schemes, but they do it so by implicit introduction of `let` $\cdot$ `in` $\cdot$ construct.

For completeness we present here a proof that the problem of closed program synthesis is PSCPACE-complete, but this is rather a folklore result, which is difficult to attribute to a particular publication. The proof is done by reduction of the type inhabitation problem for the simply-typed lambda calculus, which is known to be PSCPACE-complete [21]. Actually, the work of Augustsson [1] is based on the same observation we explicate here.

▶ **Lemma 7.**

1. *If $\Gamma \vdash_{\to} M : A$ then $\Gamma \vdash_{\mathrm{ML}} M : A$.*
2. *If $\Gamma \vdash_{\mathrm{ML}} M : A$ where Γ does not contain type schemes and $M = \mathrm{NF}(M)$ then $\Gamma \vdash_{\to} M : A$.*

Proof. The first claim follows easily by induction over M and observation that the simply-typed lambda calculus is a subsystem of ML.

The second claim follows by observation that a normal form of an ML term does not contain occurrences of `let` $\cdot$ `in` $\cdot$. As the (`let`) rule can only be applied to a term of the form `let` $\cdot$ `in` $\cdot$, this rule cannot occur in the derivation. The remaining rules are the rules of the simply-typed lambda calculus so the claim follows. ◀

▶ **Theorem 8.** *The closed program synthesis for* ML *is PSPACE-complete. This holds even for programs with algebraic types.*

Proof. By subject reduction and strong normalisation properties we may restrict our search in the closed program synthesis problem for ML to terms in normal form.

Suppose we are given a type A in the simply-typed lambda calculus. Since simply-typed lambda calculus is a subsystem of ML, we can use an algorithm for the closed program synthesis for ML on this input. In case the algorithm answers positively, there is a term M such that $\vdash_{\mathrm{ML}} M : A$. By the strong normalisation property, we may assume M is in normal form. By Lemma 7(2) we obtain $\vdash_{\to} M : A$. As a result, the answer for ML is correct for the simply-typed lambda calculus. In case the algorithm answers negatively, i.e. there is no ML term of type A, we cannot have a term of the simply-typed lambda calculus of the type either. In case there is a term M such that $\vdash_{\to} M : A$, we immediately translate M to ML by Lemma 7(1) and obtain contradiction with the correctness of the algorithm for ML. This gives a polynomial time reduction of the inhabitation problem for the simply-typed lambda calculus to the program synthesis for ML. As the inhabitation problem is PSPACE-hard for this calculus [21], we obtain that closed program synthesis for ML is PSPACE-hard.

A similar argument proves that the program synthesis problem for ML is reduced to the inhabitation problem for the simply-typed lambda calculus. As a result, we obtain that the synthesis problem is in PSPACE, which concludes the proof that the problem is PSPACE-complete.

The algebraic types do not invalidate the argument as they only serve as new atoms. ◀

3.1 Program Synthesis in Context

The above-discussed version of the program synthesis problem does not cover the issue of program synthesis in full. Actually, programmers do not want their programs to be composed entirely from scratch. Instead they want to use a libraries with functions that offer more refined functionality than the basic language. This consideration leads to the following version of the program synthesis problem for ML.

▶ **Problem 9** (Program Synthesis for ML)**.** *Given a context* Γ *and type* A *of* ML *check if there is a term* M *such that* $\Gamma \vdash_{\mathrm{ML}} M : A$.

One may be curious why the context Γ is not restricted to contain only types that are inhabited within the original type theory, i.e. ML in this case — procedures in a library must have been written in the same language. This broadening of the scope of possible contexts has two reasons. First, languages such as Haskell or Ocaml use, as mentioned in the introductory Section 1, page 3, extensions that make it possible to go beyond ML type discipline. Second, the libraries may be the result of incorporating some libraries written in foreign languages, e.g. Qt or GTK, which may use some internal global state and in this way enable presence of originally non-inhabited types.

One can observe that the synthesis problem of this kind was posed before in terms of Hilbert-style propositional logic. In such systems axiom schemes may be regarded as polymorphic operations that are available in the context. The provability problem (that may be viewed as the synthesis problem through the Curry-Howard isomorphism) is undecidable there, which is stated in the Linial-Post theorem [11] that holds even for the calculae with arrow only [19], which are very close to ML. However, these Hilbert-style systems use only two rules, namely the *modus ponens* together with the substitution rule and these are not enough to guarantee that the deduction theorem holds. Therefore, these results cannot be

applied directly to the case of ML and we give here a direct reduction of the halting problem for two-counter automata.

A *two-counter automaton* $\mathcal{A}$ (introduced by Minsky [13]) is a tuple $\langle Q, q_I, F, \delta\rangle$ where Q is a finite set of states, $q_I \in Q$ is an initial state, $F \subseteq Q$ is a set of final states, and δ is a set of rules of the form

$$q, T \mapsto q', k_0, k_1 \tag{1}$$

where $q, q' \in Q$, $T \in \{Z_0, Z_1, NZ_0, NZ_1\}$, $k_0, k_1 \in \{0, -1, +1\}$. The value T describes a test on one of the available two counters that enables the rule to fire. It can be described precisely by means of configurations. A *configuration* of the automaton is a triple $\langle q, l_0, l_1\rangle$ where $q \in Q$, $l_0, l_1 \in \mathbb{N}$. A rule (1) is applicable to a configuration $\langle q, l_0, l_1\rangle$ when

- $T = Z_i$, $l_i = 0$, $l_{\bar{i}} \neq 0$, and $k_i \geq 0$, or
- $T = Z_i$, $l_i = 0$, $l_{\bar{i}} = 0$, and $k_0, k_1 \geq 0$, or
- $T = NZ_i$, $l_i \neq 0$, and $l_{\bar{i}} \neq 0$, or
- $T = NZ_i$, $l_i \neq 0$, $l_{\bar{i}} = 0$, and $k_{\bar{i}} \geq 0$

for $i \in \{0, 1\}$ and $\bar{i} = i + 1 \mod 2$. Observe that the rules that perform subtraction can only be fired when the resulting counter is non-negative. We fix an automaton $\mathcal{A}$ for the rest of the section. Halting is defined inductively as follows. We say that the automaton *halts from a configuration* $\langle q, l_0, l_1\rangle$ when

- either $q \in F$ or
- there is a rule $q, T \mapsto q', k_0, k_1$ applicable to $\langle q, l_0, l_1\rangle$ such that $\mathcal{A}$ halts from $\langle q', l_0 + k_0, l_1 + k_1\rangle$.

We say that the automaton *halts* when it halts from the configuration $\langle q_I, 0, 0\rangle$.

In our construction we use the following vocabulary of type constants and constructors

- type constants: $\mathsf{loop}, \mathsf{start}, p, 1, 0$ and
- type constructors: P, R of arity 1.

We need first to provide representation for states. For this we define types $R^0(A) = A$ and $R^{i+1}(A) = R(R^i(A))$. Assume that the set of states is $Q = \{q_0, \ldots, q_n\}$. We can represent the state q_i by $A_{q_i} = R^i(0)$. We write $L_0^0 = 0, L_1^0 = 1, L_0^{i+1} = P(P^i(0)), L_1^{i+1} = P(P^i(1))$. We can now define an ML type $A\langle q, l_0, l_1\rangle$ that represents the configuration $\langle q, l_0, l_1\rangle$ of the automaton

$$A\langle q, l_0, l_1\rangle = L_0^{l_0} \to L_1^{l_1} \to A_q \to p. \tag{2}$$

With this in mind, we can define formulas that represent particular kinds of automaton rules. Let us define first formulas that make it possible to test for zero and non-zero:

$$B_{Z,0}(D_1, D_2) = L_0^0 \to D_1 \to D_2 \to p, \qquad B_{Z,1}(D_1, D_2) = D_1 \to L_1^0 \to D_2 \to p,$$
$$B_{NZ,0}(D_1, D_2, D_3) = P(D_1) \to D_2 \to D_3 \to p, \quad B_{NZ,1}(D_1, D_2, D_3) = D_1 \to P(D_2) \to D_3 \to p$$

as well as the formula that updates counters

$$C(\alpha, \beta, \gamma) = \alpha \to \beta \to \gamma \to p.$$

We can now define operations $B + k$ for $k \in \mathbb{Z}$.

$$B + 0 = B, \quad B + (k + 1) = P(B) + k \text{ for } k \geq 0$$
$$P(B) + (k - 1) = B + k, \text{ for } k < 0$$
$$B + k = B, \text{ for } k < 0 \text{ and } B \neq P(B')$$

We define $B-k = B+(-k)$. Observe that for $n=-1$ we have $\gamma - n = P(\gamma)$ and $\gamma - n + n = \gamma$. The context $\Gamma_{\mathcal{A}}$ consists of the following six kinds of type schemes:

(1) $(A\langle q_I, 0, 0\rangle \to \mathsf{loop}) \to \mathsf{start}$
(2) for each rule of the form $q, Z_0 \mapsto q', m, n$:
$\forall\gamma. B_{Z,0}(\gamma - n, A_q) \to (C(L_0^0 + m, \gamma - n + n, A_{q'}) \to \mathsf{loop}) \to \mathsf{loop}$
(3) for each rule of the form $q, Z_1 \mapsto q', m, n$:
$\forall\gamma. B_{Z,1}(\gamma - m, A_q) \to (C(\gamma - m + m, L_1^0 + n, A_{q'}) \to \mathsf{loop}) \to \mathsf{loop}$
(4) for each rule of the form $q, NZ_0 \mapsto q', m, n$:
$\forall\gamma_0\gamma_1. B_{NZ,0}(\gamma_0, \gamma_1 - n, A_q) \to$
$(C(\gamma_0 + 1 + m, \gamma_1 - n + n, A_{q'}) \to \mathsf{loop}) \to \mathsf{loop}$
(5) for each rule of the form $q, NZ_1 \mapsto q', m, n$:
$\forall\gamma_0\gamma_1. B_{NZ,1}(\gamma_0 - m, \gamma_1, A_q) \to$
$(C(\gamma_0 - m + m, \gamma_1 + 1 + n, A_{q'}) \to \mathsf{loop}) \to \mathsf{loop}$
(6) $\forall\gamma_0\gamma_1.(\gamma_0 \to \gamma_1 \to A_q \to p) \to \mathsf{loop}$ for $q \in F$

where $k_0, k_1 \geq 0$. An important property of $\Gamma_{\mathcal{A}}$ is that all its type schemes have targets being type constants. Since the notation above is quite dense, we give an example on how it expands. Suppose we want to obtain the concrete formula for the rule $q_1, Z_0 \mapsto q_2, +1, -1$. Assume that q_1 is represented as $R(\mathsf{0})$ and q_2 as $R(R(\mathsf{0}))$. The rule falls under the point (2) above and gives rise to the formula

$$\forall\gamma.(\mathsf{0} \to P(\gamma) \to R(\mathsf{0}) \to p) \to ((P(\mathsf{0}) \to \gamma \to R(R(\mathsf{0})) \to p) \to \mathsf{loop}) \to \mathsf{loop}.$$

It is worth pointing out here that the mentioned above type schemes are variations on the double negation principle (they turn into real double negation when loop is understood as falsity) and in this way resemble the very natural terms presented in our example on pages 12:1–12:2.

We say that a context Σ is *faithful for the automaton* $\mathcal{A}$ when

- it consists only of pairs in one of the form: $x : A\langle q, l_0, l_1\rangle$,
- if $x : A\langle q, l_0, l_1\rangle \in \Sigma$ then either $q = q_I, l_0 = 0, l_1 = 0$ or there is a rule $q', T \mapsto q, k_0, k_1$ in δ, and a pair $x : A\langle q', l_0', l_1'\rangle \in \Sigma$ such that the rule is applicable to $\langle q', l_0', l_1'\rangle$, $l_0 = l_0' + k_0$, and $l_1 = l_1' + k_1$.

▶ **Lemma 10.** *If Σ is faithful for the automaton $\mathcal{A}$ with $x : A\langle q, l_0, l_1\rangle \in \Sigma$ such that $\mathcal{A}$ halts from $\langle q, l_0, l_1\rangle$ then there is a term M such that $\Gamma_{\mathcal{A}}, \Sigma \vdash_{\mathrm{ML}} M : \mathsf{loop}$.*

Proof. The proof is by a straightforward induction over the notion of halting from configuration. ◀

▶ **Lemma 11.** *If Σ is faithful for the automaton $\mathcal{A}$ and $\Gamma_{\mathcal{A}}, \Sigma \vdash_{\mathrm{ML}} M : \mathsf{loop}$ for some term M then there is $x : A\langle q, l_0, l_1\rangle \in \Sigma$ such that $\mathcal{A}$ halts from $\langle q, l_0, l_1\rangle$.*

Proof. We may assume that M is in normal form. The proof is by induction on the size of the term M. There is no term of size 1 of type loop so the base case follows.

For the inductive step we observe that M is in normal form and its type is a constant so it must be of the form $xM_1 \dots M_n$ for some $x : \tau \in \Gamma_{\mathcal{A}}, \Sigma$. As no type in Σ has target loop we obtain that $x : \tau \in \Gamma_{\mathcal{A}}$. We analyse the cases for possible kinds of schemes in $\Gamma_{\mathcal{A}}$. We present here only the most interesting cases (2) and (4).

In case (2), $x : \forall\gamma. B_{Z,0}(\gamma - n, A_q) \to (C(L_0^0 + m, \gamma - n + n, A_{q'}) \to \mathsf{loop}) \to \mathsf{loop}$ for a transition rule $q, Z_0 \mapsto q', m, n$ in δ (*). This means that $M = xM_1M_2$ where M_1 is of type $B_{Z,0}((\gamma - n)[\gamma := A_0], A_q)$, the term M_2 is of type $C(L_0^0 + m, (\gamma - n + n)[\gamma := A_0], A_{q'}) \to \mathsf{loop}$.

As $B_{Z,0}((\gamma - n)[\gamma := A_0], A_q) = L_0^0 \to A_2 \to A_q \to p$, the term $M_1 = \lambda x_1 x_2 x_3.M_1'$ where $\Gamma_{\mathcal{A}}, \Sigma, x_1 : L_0^0, x_2 : A_2, x_3 : A_q \vdash_{\mathrm{ML}} M_1' : p$ with $A_2 = (\gamma - n)[\gamma := A_0]$. Only elements of Σ have target p so $M_1' = yM_1''M_2''M_3''$ for some $y : A\langle q_y, l_0, l_1\rangle = L_0^{l_0} \to L_1^{l_1} \to A_{q_y} \to p$, M_1'' of type $L_0^{l_0}$, M_2'' of type $L_1^{l_1}$ and M_3'' of type A_{q_y}. The only type in the context that has target of the form $P(\cdots P(0)\cdots)$ is the type of x_1 so $M_1'' = x_1$. Similarly, the only type in the context that has target of the form $P(\cdots P(1)\cdots)$ is the type of x_2 so $M_2'' = x_2$. Again, the only type in the context that can possibly have the target of the form $R(\cdots R(0)\cdots)$ is the type of x_3, so $M_3'' = x_3$ and $A_2 = (\gamma - n)[\gamma := A_0] = R(\cdots R(0)\cdots)$. Consequently $y : A\langle q, 0, l_1\rangle \in \Sigma$ for some number $l_1 \geq 0$. Recall that $\gamma - n$ for $n = -1, 0, 1$ is either γ or $P(\gamma)$ so γ is instantiated either with $A_0 = P^{l_1}(1)$ or $A_0 = P^{l_1-1}(1)$ (for $n = -1$). We can now turn our attention to the term M_2. Since its type is $A_g \to \mathsf{loop} = C(L_0^0 + m, (\gamma - n + n)[\gamma := A_0], A_{q'}) \to \mathsf{loop}$, it has the form $\lambda x_1.M_2'$ where $\Gamma, \Sigma, x_1 : A_g \vdash_{\mathrm{ML}} M_2' : \mathsf{loop}$. An analysis of A_g shows that it is actually $A\langle q', 0 + m, l_1 + n\rangle$. As the size of M_2' is less than the size of M, we can apply the induction hypothesis. As a result, we obtain $A\langle q'', k_0, k_1\rangle \in \Sigma$ such that $\mathcal{A}$ halts from $\langle q'', k_0, k_1\rangle$. In case $\langle q'', k_0, k_1\rangle = \langle q', 0 + m, l_1 + n\rangle$, we obtain by the mentioned above rule (*) that $\mathcal{A}$ halts from $\langle q, 0, l_1\rangle$ where $A\langle q, 0, l_1\rangle \in \Sigma$. In case $\langle q'', k_0, k_1\rangle \neq \langle q', 0 + m, l_1 + n\rangle$, we obtain that already $\langle q'', k_0, k_1\rangle \in \Sigma$. In both cases the claim of the current lemma is proved.

In case (4), $x : B_{NZ,0}(\gamma_0, \gamma_1 - n, A_q) \to (C(\gamma_0 + 1 + m, \gamma_1 - n + n, A_{q'}) \to \mathsf{loop}) \to \mathsf{loop}$ for a transition rule $q, NZ_0 \mapsto q', m, n$ in δ (**). This means that $M = xM_1M_2$ where M_1 is of type

$$B_{NZ,0}(\gamma_0[\gamma_0 := A_0], (\gamma_1 - n)[\gamma_1 := A_1], A_q),$$

and the term M_2 is of type

$$C((\gamma_0 + 1 + m)[\gamma_0 := A_0], (\gamma_1 - n + n)[\gamma_1 := A_1], A_{q'}) \to \mathsf{loop}.$$

As $B_{NZ,0}(\gamma_0[\gamma_0 := A_0], (\gamma_1 - n)[\gamma_1 := A_1], A_q) = P(A_0) \to (\gamma_1 - n)[\gamma_1 := A_1] \to A_q \to p$, the term $M_1 = \lambda x_1 x_2 x_3.M_1'$ where $\Gamma_{\mathcal{A}}, \Sigma, x_1 : P(A_0), x_2 : (\gamma_1 - n)[\gamma_1 := A_1], x_3 : A_q \vdash_{\mathrm{ML}} M_1' : p$. Only elements of Σ have target p so $M_1' = yM_1''M_2''M_3''$ for some $y : A\langle q_y, l_0, l_1\rangle = L_0^{l_0} \to L_1^{l_1} \to A_{q_y} \to p$ with $l_0, l_1 \geq 0$, M_1'' of type $L_0^{l_0}$, M_2'' of type $L_1^{l_1}$ and M_3'' of type A_{q_y}. The only type in the context that has target of the form $P(\cdots P(0)\cdots)$ is the type of x_1 so $M_1'' = x_1$. Similarly, the only type in the context that has target of the form $P(\cdots P(1)\cdots)$ is the type of x_2 so $M_2'' = x_2$. Again, the only type in the context that can possibly have the target of the form $R(\cdots R(0)\cdots)$ is the type of x_3, so $M_3'' = x_3$. Consequently $y : A\langle q, l_0, l_1\rangle \in \Sigma$, and $l_0 > 0$. Recall that $\gamma_1 - n$ for $n = -1, 0, 1$ is either γ_1 or $P(\gamma_1)$ so γ_1 is instantiated either with $A_1 = P^{l_1}(1)$ or $A_1 = P^{l_1-1}(1)$ (for $n = -1$). We can now turn our attention to the term M_2. Since its type is

$$A_g \to \mathsf{loop} = C((\gamma_0 + 1 + m)[\gamma_0 := A_0], (\gamma_1 - n + n)[\gamma_1 := A_1], A_{q'}) \to \mathsf{loop},$$

it has the form $\lambda x_1.M_2'$ where

$$\Gamma, \Sigma, x_1 : A_g \vdash_{\mathrm{ML}} M_2' : \mathsf{loop}.$$

An analysis of A_g shows that it is actually $A\langle q', l_0 + m, l_1 + n\rangle$. As the size of M_2' is less than the size of M, we can apply the induction hypothesis. As a result, we obtain $A\langle q'', k_0, k_1\rangle \in \Sigma$ such that $\mathcal{A}$ halts from $\langle q'', k_0, k_1\rangle$. In case $\langle q'', k_0, k_1\rangle = \langle q', l_0 + m, l_1 + n\rangle$, we obtain by the mentioned above rule (**) that $\mathcal{A}$ halts from $\langle q, l_0, l_1\rangle$ where $A\langle q, l_0, l_1\rangle \in \Sigma$. In case $\langle q'', k_0, k_1\rangle \neq \langle q', l_0 + m, l_1 + n\rangle$, we obtain that already $A\langle q'', k_0, k_1\rangle \in \Sigma$. In both cases the claim of the current lemma is proved. ◀

We can now use these lemmas to prove the undecidability result.

▶ **Theorem 12.** *The program synthesis for* ML *with algebraic types is undecidable.*

Proof. We reduce the halting problem for two-counter automata to the problem of program synthesis. Given an automaton $\mathcal{A}$ we generate the instance $\Gamma_{\mathcal{A}}, \mathsf{start}$ of the program synthesis for ML with algebraic types.

In case there is a (normal form) term M such that $\Gamma_{\mathcal{A}} \vdash_{\mathrm{ML}} M : \mathsf{start}$ we proceed as follows. Since start is atomic, the term M must be of the form $xM_1 \cdots M_n$ for some $x \in \Gamma_{\mathcal{A}}$. The only variable of this kind is $x : (A\langle q_I, 0, 0\rangle \to \mathsf{loop}) \to \mathsf{start}$ and $M = x(\lambda y.M')$ where $\Gamma_{\mathcal{A}}, y : A\langle q_I, 0, 0\rangle \vdash_{\mathrm{ML}} M' : \mathsf{loop}$. We observe that $\Sigma = y : A\langle q_I, 0, 0\rangle$ is faithful for the automaton $\mathcal{A}$. We can apply now Lemma 11 and conclude that $\mathcal{A}$ halts from $\langle q_I, 0, 0\rangle$.

In case $\mathcal{A}$ halts from $\langle q_I, 0, 0\rangle$, we can apply Lemma 10 to the context $\Sigma = y : A\langle q_I, 0, 0\rangle$, which is faithful for the automaton $\mathcal{A}$. As a result, we obtain a term M' such that $\Gamma_{\mathcal{A}}, y : A\langle q_I, 0, 0\rangle \vdash_{\mathrm{ML}} M' : \mathsf{loop}$. This gives us that $\Gamma_{\mathcal{A}} \vdash_{\mathrm{ML}} x(\lambda y.M') : \mathsf{start}$ where $x : (A\langle q_I, 0, 0\rangle \to \mathsf{loop}) \to \mathsf{start} \in \Gamma_{\mathcal{A}}$.

This concludes the reduction and thus the program synthesis for ML with algebraic types is undecidable as the halting problem is. ◀

Algebraic types in functional programming languages use constants that construct values of algebraic types (e.g. `cons` for lists) and destruct them (e.g. fold-like iterators for lists). The construction above does not work when the context Γ above contains constructors and destructors for algebraic types. However, it can be easily corrected when we allow value constructors, but disallow destructors. The proof breaks when we say that the only way to obtain target type of the forms $P^i(0), P^i(1), R^i(0)$ is through the use of a variable introduced through one of our six type schemes. In the presence of value constructors we have another option to construct them using them. We can forbid this possibility by taking a slightly different encoding of counters, and states and take there $P^i(0) \to q, P^i(1) \to q, R^i(0) \to q$ for some new type constant q.

It is still possible to have realistic programming scenarios within these constraints. Algebraic types are often hidden in abstract types that make available value constructors, but do not offer destructors. Instead they give the programmers an interface to operate on constructed values without the knowledge of their actual representation.

3.2 Program Synthesis with Restricted Instantiation

Since the problem in its full generality is undecidable we can try to find a reasonable special case, for which the problem becomes decidable. One of the critical features of the reduction in the previous section is the possibility to instantiate type schemes with types of arbitrary size (they are used to match the values of the counters during a run of a simulated automaton). Therefore, we propose to restrict the instantiation.

▶ **Problem 13** (Program Synthesis for ML_1)**.** *Given a context* Γ *and type* A *of* ML_1 *check if there is a term* M *such that* $\Gamma \vdash_{\mathrm{ML}_1} M : A$.

In this paper we show that this problem is EXPSPACE-complete for ML with algebraic types. As the main device to prove this, we use the results for the provability problem in IFOL in the restricted class $\mathbf{N}$ of first-order formulas of Mints-hierarchy [17].

We can now define a transformation $\lfloor\cdot\rfloor$ that transforms a formula from $\mathbf{B}$ to an ML type scheme. For simplicity, we assume that the set of predicates and algebraic type constructors are the same, i.e. *Preds* = *TAlg*. We assume that there is an injective correspondence between

variables in $Vars_{IFOL}$ and type variables in $TVars$. The result of this correspondence for a variable X is written α_X. On other formulas of $\mathbf{B}$ it gives

$$\lfloor P(X_1,\dots,X_n)\rfloor = P(\alpha_{X_1},\dots,\alpha_{X_n}),\quad \lfloor\varphi_1\to\varphi_2\rfloor = \lfloor\varphi_1\rfloor\to\lfloor\varphi_2\rfloor,\quad \lfloor\forall X\,\varphi\rfloor = \forall\alpha_X.\lfloor\varphi\rfloor.$$

This definition naturally extends to contexts. We can also define its extension to formulas of $\mathbf{N}$. For a formula $\varphi = \varphi_1\to\cdots\to\varphi_n\to P(X_1,\dots,X_n)$ we define (with a slight abuse of the target language) the type

$$\lfloor\varphi\rfloor = \lfloor\varphi_1\rfloor\to\cdots\to\lfloor\varphi_n\rfloor\to\lfloor P(X_1,\dots,X_n)\rfloor.$$

(Note that this is actually a System F type the inhabitation of which is equivalent to the inhabitation of the type $\lfloor P(X_1,\dots,X_n)\rfloor$ in the initial context extended with $x_1 : \lfloor\varphi_1\rfloor,\dots,\lfloor\varphi_n\rfloor$.) Committing another small abuse of notation, we define $\lfloor\cdot\rfloor$ on proof terms.

$$\lfloor x\rfloor = x,\ \lfloor\lambda x:\varphi.M\rfloor = \lambda x.\lfloor M\rfloor,\ \lfloor M_1M_2\rfloor = \lfloor M_1\rfloor\lfloor M_2\rfloor,\ \lfloor\lambda X\,M\rfloor = M,\ \lfloor MX\rfloor = \lfloor M\rfloor.$$

Note that this translation may be viewed as a formula erasure mapping.

Here are the basic properties of the transformation.

▶ **Lemma 14.**
1. *For each formula ϕ and variables $X,Y\in Vars_{IFOL}$ $\lfloor\phi[X:=Y]\rfloor = \lfloor\phi\rfloor[\alpha_X:=\alpha_Y]$.*
2. *For each proof term M in normal form the term $\lfloor M\rfloor$ is also in normal form.*

Proof. Straightforward induction over the formula ϕ in (1) and term M in (2). ◀

▶ **Lemma 15.** *For each Γ with formulae from $\mathbf{\Pi}_1\cap\mathbf{B}$ only and $\varphi\in\mathbf{O}$ it holds that for each proof term M in normal form $\Gamma\vdash_{\mathrm{IFOL}} M:\varphi$ if and only if $\lfloor\Gamma\rfloor\vdash_{\mathrm{ML_1}}\lfloor M\rfloor:\lfloor\varphi\rfloor$.*

Proof.
(⇒) The proof is by induction over the derivation for $\Gamma\vdash_{\mathrm{IFOL}} M:\varphi$ with cases depending on the last rule used. The interesting case is when the last rule is $(\forall E)$ then $M = xX_1\cdots X_m$ and the judgement $\Gamma\vdash xX_1\cdots X_m:\varphi$ as well as $\Gamma\vdash x:\forall Y_1\dots Y_m.\varphi[X_1:=Y_1,\dots X_m:=Y_m]$ are derivable in intuitionistic first-order logic. This means that $x:\forall Y_1\dots Y_m.\varphi[X_1:=Y_1,\dots X_m:=Y_m]\in\Gamma$. Observe that $\lfloor\forall Y_1\dots Y_m.\varphi[X_1:=Y_1,\dots,X_m:=Y_m]\rfloor = \forall\alpha_{Y_1}\dots\alpha_{Y_m}.\lfloor\varphi[X_1:=Y_1,\dots,X_m:=Y_m]\rfloor$ and consequently that

$$x:\forall\alpha_{Y_1}\dots\alpha_{Y_m}.\lfloor\varphi[X_1:=Y_1,\dots,X_m:=Y_m]\rfloor\in\lfloor\Gamma\rfloor.$$

We can now use the (var) rule of $\mathrm{ML_1}$ with the instantiation $S = [\alpha_{Y_1}:=\alpha_{X_1},\dots,\alpha_{Y_m}:=\alpha_{X_m}]$ since this instantiation can be used to obtain

$$S(\lfloor\varphi[X_1:=Y_1,\dots,X_m:=Y_m]\rfloor)\ \mathsf{inst}_1\ \forall\alpha_{Y_1}\dots\alpha_{Y_m}.\lfloor\varphi[X_1:=Y_1,\dots,X_m:=Y_m]\rfloor.$$

As a result, we obtain $\lfloor\Gamma\rfloor\vdash_{\mathrm{ML_1}} x : S(\lfloor\varphi[X_1:=Y_1,\dots,X_m:=Y_m]\rfloor)$, which is actually $\lfloor\Gamma\rfloor\vdash_{\mathrm{ML_1}} x:\lfloor\varphi\rfloor$ as $\lfloor\varphi\rfloor = S(\lfloor\varphi[X_1:=Y_1,\dots,X_m:=Y_m]\rfloor)$.

(⇐) The proof is by induction over derivation for $\lfloor\Gamma\rfloor\vdash_{\mathrm{ML_1}}\lfloor M\rfloor:\lfloor\varphi\rfloor$. The interesting case is when the last rule is (var) then our judgement has the form $\lfloor\Gamma\rfloor, x:\lfloor\psi\rfloor\vdash x:\lfloor\varphi\rfloor$ for some ψ, and it holds that $\lfloor\psi\rfloor\ \mathsf{inst}_1\ \lfloor\varphi\rfloor$. Since $\psi\in\mathbf{\Pi}_1\cap\mathbf{B}$ and $\varphi\in\mathbf{O}$, the formula ψ is $\forall X_1\dots X_n.\psi_0$, where ψ_0 is quantifier-free and φ is quantifier-free. This implies that $\lfloor\psi\rfloor = \forall\alpha_{X_1}\dots\alpha_{X_n}.\lfloor\psi_0\rfloor$. Moreover, relation inst_1 means that $\lfloor\psi_0\rfloor[\alpha_{X_1}:=\beta_1,\dots,\alpha_{X_n}:=\beta_n] = \lfloor\varphi\rfloor$ where $\beta_1,\dots,\beta_n$ are type variables. For those of the variables that occur in $\mathrm{FTV}(\lfloor\varphi\rfloor)$ we may assume

that are of the form α_Y. For other ones we can also assume they have analogous form. Therefore, we have $\lfloor\psi_0\rfloor[\alpha_{X_1} := \alpha_{Y_1}, \ldots, \alpha_{X_n} := \alpha_{Y_n}] = \lfloor\varphi\rfloor$ for some $Y_1, \ldots, Y_n \in \mathit{Vars}_{IFOL}$. We can now use the (var) rule of intuitionistic first-order logic to obtain $\Gamma, x : \psi \vdash x : \psi$ and then subsequently n times the $(\forall E)$ rule with substitutions of the form $[X_i := Y_i]$ to obtain $\Gamma, x : \psi \vdash xY_1 \cdots Y_n : \psi_0[X_1 := Y_1, \ldots, X_n := Y_n]$, which is the required judgement as $\lfloor\psi_0[X_1 := Y_1, \ldots, X_n := Y_n]\rfloor = \lfloor\psi_0\rfloor[\alpha_{X_1} := \alpha_{Y_1}, \ldots, \alpha_{X_n} := \alpha_{Y_n}]$ by Lemma 14 and $\lfloor xY_1 \cdots Y_n\rfloor = x$ by definition of $\lfloor\cdot\rfloor$. ◀

The lemma above helps in giving the proof of EXPSPACE-hardness. One might be tempted to give a proof that the problem is in EXPSPACE also through a translation argument. However, such reasoning would be complicated as the arguments of type constructors need not be type variables. Therefore, we present a direct proof.

First, we need to know how type substitutions affect inferences in ML.

▶ **Lemma 16.** *If* $\Gamma \vdash N : A$ *in either* ML *or* ML_1 *then* $\Gamma[\vec{\alpha} := \vec{\beta}] \vdash N : A[\vec{\alpha} := \vec{\beta}]$ *in* ML *or* ML_1 *respectively.*

Proof. Straightforward induction over the term N. ◀

▶ **Lemma 17.** *Let* $\mathcal{C}$ *be the set of type constants that occur in* Γ, A*. If* N *is in normal form and* $\Gamma \vdash N : A$ *in* ML_1 *where* $\mathcal{C} \cup \mathrm{FTV}(\Gamma, A) \subseteq \mathcal{V}$ *for some non-empty set* $\mathcal{V}$ *then all type instantiations in the derivation can be restricted to use atoms in* $\mathcal{V}$*.*

Proof. The proof is by induction over the term N.

In case N is a variable the proof is obvious. In case $N = \lambda x.N'$, the type A is $A_1 \to A_2$ and the inference must end with the $(\to I)$ rule. This reduces the problem to the one for the judgement $\Gamma, x : A_1 \vdash N' : A_2$, for which we can apply the induction hypothesis and then obtain our conclusion.

In case $N = xN_1 \ldots N_k$, the inference must start with the (var) rule for $x : \sigma \in \Gamma$ where $\sigma = \forall\vec{\gamma}.A_1 \to \cdots \to A_k \to B$ and $\Gamma \vdash N_i : A_i[\vec{\gamma} := \vec{A}]$ where $\vec{A}$ are atomic. By Lemma 16 we obtain $\Gamma \vdash N_i : A_i[\vec{\gamma} := \vec{A'}]$ for $i = 1, \ldots, k$ where $\vec{A'}$ differs from $\vec{A}$ only on positions that are outside of $\mathcal{V}$ and has a fixed element α_0 of $\mathcal{V}$ there. Therefore the atomic instantiation with $\vec{A'}$ can be used in the initial (var) rule instead of $\vec{A}$, which gives the required conclusion after application of the induction hypothesis to arguments N_i for $i = 1, \ldots, k$. ◀

▶ **Lemma 18.** *The program synthesis problem for* ML_1 *is in EXPSPACE.*

Proof. Given a context Γ and type A we use a simple generalisation of the Ben-Yelles algorithm [20]. Lemma 17 implies that type inference for a normal program N of type A above can be restricted to use only type atoms from the set $\mathcal{A} = \mathcal{C} \cup \mathrm{FTV}(A, \Gamma) \cup \{\alpha_0\}$, where $\mathcal{C}$ is the set of type constants in A, Γ. Note that the type variable α_0 guarantees that $\mathcal{A}$ is not empty. Therefore the algorithm needs only to consider judgements $\Gamma' \vdash M : B$ where all type atoms are in $\mathcal{A}$. It should be clear that the number of different types in Γ' is at most exponential in the size n of A, Γ. (A type scheme that generalises m variables has at most m^n instances.) Using the same argument as for simply typed lambda calculus in the Ben-Yelles algorithm we obtain an alternating exponential time exponential algorithm. ◀

▶ **Theorem 19.** *The program synthesis problem for* ML_1 *is EXPSPACE-complete.*

Proof. We can now exploit Theorem 5 in the context of the program synthesis problem. The EXPSPACE-hardness is the result of this theorem combined with the reduction presented in Lemma 15. The fact that the problem is in EXPSPACE is again the the result of the theorem combined this time with the construction presented in Lemma 18. ◀

4 Conclusions and Further Work

We presented an initial study on program synthesis for functional programs with libraries. The goal of the constructions presented here is not only to demonstrate undecidability or a particular complexity. They also allow us to better understand intricate difficulties of program synthesis. In particular, the undecidability proof works because the depth of instantiation is not bounded. This makes it reasonable to restrict it in practical procedures or heuristics for program synthesis. We make restriction of this kind in our analysis for ML with instantiations restricted to atomic types. The EXPSPACE-hardness proof there relies on the translation from the core of the fragment Σ_1 of intuitionistic first-order logic. This translation is direct so the reason for the high complexity in the logic is the same as in ML, namely, the number of quantifiers in front of a type scheme occurs in the exponent while the number of atoms determines the base of exponentiation. This is also a hint on the heuristics for program synthesis so that they should give ways to restrict type schemes in this fashion.

The discussion of this paper goes beyond the solution of Djinn, which handles polymorphic functions, but does not allow for any instantiation. One improvement of the current work over Djinn is that we show that handling of instantiation may lead to undecidability, but also we show how to handle some interesting instantiation cases.

There are still some interesting questions that are left open here. First, it is not clear what is the complexity of program synthesis with context for ML programs with no algebraic types. Second, the complexity of program synthesis for the case where the context contains only types of the algebraic type constructors and destructors may be different than the complexities obtained in this work. It is also appealing to investigate the impact of the size of the instantiation on the complexity of the problem in the spirit of the bounded combinatory logic studied by Düdder et al [7]. These theoretical questions can be complemented by investigations in more practical directions. For example, which syntactic forms of types give rise to small number of inhabitants? What are the situations when the resulting terms are small enough to be further examined by programmers in reasonable time?

References

1 Lennart Augustsson, 2005. URL: `https://github.com/augustss/djinn`.

2 Coq Development Team. *The Coq Proof Assistant Reference Manual V8.4*, March 2012. URL: `http://coq.inria.fr/distrib/V8.4/refman/`.

3 Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, pages 95–120, 1988.

4 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, New York, NY, USA, 1982. ACM.

5 Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

6 Catherine Dubois. Proving ML type soundness within Coq. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000 Portland, OR, USA, August 14–18, 2000 Proceedings*, pages 126–144, Berlin, Heidelberg, 2000. Springer.

7 Boris Düdder, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. Bounded Combinatory Logic. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) – 26th International Workshop/21st Annual Conference of the EACSL*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 243–258, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

8 J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmetique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

9 Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

10 Pierre Letouzey. Coq Extraction, an Overview. In C. Dimitracopoulos A. Beckmann and B. Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

11 Samuel Linial and E. L. Post. Recursive unsolvability of the deducibility, Tarski's completeness, and independence of axioms problems of propositional calculus. *Bulletin of the American Mathematical Society*, vol. 55, p. 50, 1949. Abstract.

12 Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North-Holland, 1982.

13 Marvin L. Minsky. Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machines. *Annals of Mathematics*, 74(3):437–455, 1961.

14 T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Springer, 2002.

15 Christine Paulin-Mohring. Fω's programs from proofs in the calculus of constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1989.

16 J. C. Reynolds. Towards a theory of type structure. In Ehring et al., editor, *Programming Symposium, Proceedings Colloque Sur La Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

17 Aleksy Schubert, Paweł Urzyczyn, and Konrad Zdanowski. On the Mints hierarchy in first-order intuitionistic logic. In A. Pitts, editor, *Foundations of Software Science and Computation Structures 2015*, volume 9034 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2015. doi:10.1007/978-3-662-46678-0_29.

18 Helmut Schwichtenberg. The MINLOG system. URL: http://www.mathematik.uni-muenchen.de/~logik/minlog/.

19 W. E. Singletary. Many-one degrees associated with partial propositional calculi. *Notre Dame J. Formal Logic*, 15(2):335–343, 04 1974.

20 M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149. Elsevier, 2006.

21 Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.

22 Nicholas Wirth. *Systematic Programming: An Introduction*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1973.