DOI: 10.15439/2022F135
ISSN 2300-5963 ACSIS, Vol. 32

Type System of Anemone Functional Language

Paweł Batko, Marcin Kuta
AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Krakow, Poland,
Institute of Computer Science,
Faculty of Computer Science, Electronics and Telecommunications
Email: mkuta@agh.edu.pl

***Abstract*—Anemone is a functional language, which provides an actor system as its model of concurrency. This paper describes type system of the Anemone language. The type system is the strong point of Anemone. In comparison to a dynamic type system, the static type system of Anemone guarantees more exact error detection. The full type inference disposes the programmer from explicit specification of type labels. As the type system of Anemone is polymorphic, code conciseness, rich data structures and pattern matching are provided in Anemone.**

I. Introduction

ANEMONE language is a functional language [1], [2] inspired by Scala, Erlang, Haskell, and ML. It supports a wide range of features including actors communicating via messages, tight integration with the LLVM infrastructure, interoperability with the C language, and automatic memory management with garbage collector.

Anemone language is equipped with a static, polymorphic type system, with full type inference based on let-polymorphism. The static type system guarantees that more errors are detected than using a dynamic type system. Error detection is also performed earlier – at the compile time rather than the run time. The full type inference disposes the programmer from explicit specification of type labels in a program. As the type system of Anemone is polymorphic, the code is concise and algebraic data types enable rich data structures and pattern matching.

In this paper, we show the design and implementation of the type system adopted in Anemone and the inference algorithm. Type inference has been based on the Hindley-Milner algorithm. The Hindley-Milner inference has been implemented with algorithm W. In particular, we present the extension of algorithm W introduced for the purpose of the Anemone language. We also show data structures which are the basis for above algorithms, including representations of type variables and type schemes.

II. Type Systems in Functional Languages

The choice of a type system is an important decision which determines many aspects of compiler architecture. The two main options are dynamic typing and static typing.

Contrary to no typing, with dynamic typing each value is ascribed a particular type. The type correctness of a program is performed during the run time. Dynamic typing offers the programmer flexibility because he does not have to adjust his program to a static type system. The drawback of this approach is the reduced amount of static information about a compiled program. This reduces the range of applicable optimisations and imposes a generation of the additional code which verifies the program correctness with respect to its types. Dynamic typing introduces an additional overhead during the run time. Scheme and Clojure are examples of functional languages which use this approach. Anemone is not typed dynamically as lack of accurate error detection at the compile time is an important drawback of this approach.

Static typing associates a label denoting its type with each variable. The current type systems originate from the simply typed lambda-calculus [3]. The set of operations that may be performed on a given variable are limited by the type of variable that it is. Static typing offers several advantages:

- an early and exact error detection,
- increased opportunities for code optimizations due to additional static information,
- type information is a form of code documentation,
- better support for programming environments.

Type systems based on dependent typing can express types which depend not only on other types, but also on values of variables. For example, they can guarantee that a function taking number n will return a list of length n. Epigram [4] and Agda [5] implement such a type system.

III. Polymorphic Type Systems

The core concept of a polymorphic type system is to create within a language abstractions of values which are independent from type. Functions and data types are examples of such abstractions. Parametric polymorphism enables generic code typing and uses type variables, which are replaced with concrete types when needed. Other kinds of polymorphism include:

- *ad-hoc* polymorphism, which associates many implementations with a single function identifier. The proper function is chosen on the basis of passed parameters.
- subtype polymorphism, the examples of which are subtypes present in object-oriented languages.

In Figure 1, the function `map` can take a list of elements of type *int* as well as a list of elements of type *string*. In a polymorphic type system, the function only needs to be defined once. In languages which do not support polymorphism, the programmer would have to implement a distinct function

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

map id ["1", "2"]
map id [1, 2]
```

Fig. 1: Example of a polymorphic type system

The function `map` takes two parameters. The first parameter is function `f` mapping any type `a` to any type `b`. The second parameter is a list of elements of type `a`. The function `map` returns a list of elements of type `b`. Example from *Haskell*

for each of types or would have to typecast to `void*` type, sacrificing type safety, as is usually done in *C*.

Parametric polymorphic type systems further divide into predicative polymorphism and impredicative polymorphism [6].

Predicative polymorphism distinguishes monotypes from polytypes. A monotype does not have type variables, e.g., *int*, *int* $\mapsto$ *bool*. A polytype can be parametrised with a type variable, e.g. $\forall a.a$, where a is a type variable rather than a concrete type. A type variable can be substituted with any monotype.

Let us temporarily assume that a constructor of function type, $\rightarrow$, is the only available constructor of type. In rank-1 (prenex) polymorphism, a polytype may contain a universal quantifier located only at the leftmost position. This means that the quantified variable can be only substituted with a monotype, and a polytype containing universal quantifier cannot be on the left hand side of $\rightarrow$ constructor, provided that the given type is encoded as a tree.

Rank-2 polymorphism forbids the presence of a universal quantifier in the encoding of a type as a tree at positions passing through two arrows. This rule can be applied to define polymorphisms of higher ranks. For example, $\forall a.a \rightarrow a$ is both rank-1 and rank-2 type (and higher), while $(\forall a.a \rightarrow a) \rightarrow$ *bool* is only rank-2 type (and higher). The crucial caveat of rank-3 polymorphism (or higher) is undecidability of its full type reconstruction [7].

Impredicative polymorphism is the strongest form of parametric polymorphism. It can express types where a type variable can be substituted with any type or type variable. For example, given type $z = \forall a\ list[a]$, type variable, a, can by substituted with z. *Haskell* is a prominent example of a language implementing such a type system.

An example of a rank-1 polymorphic type system is let-polymorphism – this is characterised by its restriction of type generalisation to the syntactic *let* construct. Let-polymorphism is used in some versions of *ML*, and it has been implemented in Anemone. An advantage of languages which implement rank-1 polymorphism is the possibility of performing global type inference without type annotations in the source code.

Anemone language is equipped with a static type system with let-polymorphism and global type inference. This solution is much more complicated than dynamic typing. This has been chosen due to additional static guarantees of program correctness, avoiding overhead linked to type annotations (achieved with type inference) and flexibility in creating abstractions which use the polymorphism mechanism.

Table I summarizes type systems of few functional languages.

TABLE I: Type systems of different functional languages

Type system	Language
Dynamic typing	Scheme, Closure
Impredicative polymorphism	Haskell
Dependent typing	Epigram, Agda
Let-polymorphism	ML, Anemone

IV. Implementation of the Type System

Implementation of the static, polymorphic type system is based on the unification algorithm and algorithm W [8].

A. Unification algorithm

The type inference algorithm in Anemone applies the unification algorithm [9]. Unification searches for substitutions of values for variables, such that two unified expressions become equal.

Let us consider expressions $f(a, x)$ and $f(y, f(y, b))$, where a and b denote values, and x and y are variables. To unify them, substitution $S = [a/y, f(a, b)/x]$ should be applied, which denotes that y should be substituted with a, and x should be substituted with $f(a, b)$. The application of a substitution to an expression is written as: $[a/y, f(a, b)/x]f(a, x)$. Substitution S is a composition of two substitutions, $S = [f(y, b)/x] \circ [a/y]$. Substitution S is called a unifier of $f(a, x)$ and $f(y, f(y, b))$, because after its application, these expressions become equal. In the context of the implemented type inference, variables refer to type variables, values refer to types, and functions refer to type constructors.

The idea of the unification algorithm is to find the principal unifier (the most general unifier) of two expressions. This is such a unifier U, that any unifier S of two expressions can be constructed by the composition of some unifier T with the principal unifier U, written as $S = U \circ T$.

The Anemone compiler implements the unification algorithm (Fig. 2) which operates on type variables and types. Figure 3 presents data types which are defined in the Anemone compiler and are needed by the unification algorithm. Type *TypeVar* represents type variables, and type *TypeApp* represents simple and complex types. For example, variable x would be represented as `TypeVar("x")`, and type constructor $f(x, a)$ (where a is a type) as `TypeApp("f", TypeVar("x"), TypeApp("a"))`.

B. Hindley-Milner type inference

Hindley-Milner type inference [8] [10] is defined for two languages: the language of expressions L_e and the language of types L_t. The language of expressions, defined with grammar in Fig. 4a, contains variables x, expressions e, function definitions, function calls and *let* expressions.

The language of types, generated by grammar in Fig. 4b, defines types of these expressions. It defines primitive types ι,

```
function Unify(s, t):
    if $t = x$ and $s = y$ then
        $[x/y]$
    else if $s = x$ and not Occurs (x, t) then
        $[t/x]$
    else if $t = x$ and not Occurs (x, s) then
        $[s/x]$
    else if $s = f(p_1, p_2, .., p_n)$ and $t = f(p'_1, p'_2, .., p'_n)$ then
        $U_1$ = Unify($p_1$, $p'_1$)
        Unify($f(U_1 p_2, .., U_1 p_n)$, $f(U_1 p'_2, .., U_1 p'_n)$) $\circ$ $U_1$
    else
        Fail
    end
function Occurs(x, t):
    if $t = f(p_1, p_2, .., p_n)$ then
        if $\exists i.\ x = p_i$ then
            true
        else
            $\exists i.$ Occurs($x$,$p_i$)
        end
    else
        false
    end
```

Fig. 2: Unification algorithm presented with function *Unify*

Function *Unify* takes two type schemes, *s*, *t*, (i.e., types, which may contain a universal quantifier) as its parameters and returns their principal unifier, if it exists. Function *Occurs* takes type variable x and type scheme t as its parameters, and checks, whether variable x occurs in the definition of type scheme *t*.

```
sealed trait TypeTerm

case class TypeVar(name: String)
extends TypeTerm

case class TypeApp(name: String,
                   typeTerms: Seq[TypeTerm]
                             = Seq.empty)
extends TypeTerm
```

Fig. 3: Implementation of variables and types values in the Anemone compiler

type variables α, type expressions τ, and type schemes σ. Type expressions τ describe types without a universal quantifier, whereas type scheme σ also describes types with a universal quantifier. Universally quantified type variables are referred to as generic type variables. The remaining type variables are free variables.

Inference rules are written as:

$$A \vdash e : \sigma\,,$$

which means that under assumption A, expression e has a type defined with type scheme σ. Symbol A_x denotes the set of hypotheses concerning types from A without hypotheses concerning x.

$$e ::= x \mid ee' \mid \lambda x.e \mid \text{let } x = e \text{ in } e'$$

(a) Grammar of the language of expressions L_e

$$\sigma ::= \tau \mid \forall \alpha \sigma$$
$$\tau ::= \alpha \mid \iota \mid \tau \to \tau$$

(b) Grammar of the language of types L_t

$$\begin{aligned} e ::= {} & x \\ & \mid e(e_1, e_2, .., e_n) \\ & \mid \lambda(x_1, x_2, .., x_n).e \\ & \mid \text{let } x = e \text{ in } e' \\ & \mid \mathit{fix}(x, e) \\ & \mid \mathit{noarg}(e) \end{aligned}$$

(c) Grammar of the language of expressions used in the compiler of *Anemone*

Fig. 4: Grammar of languages used in the Hindley-Milner inference

Type schemes can be ordered w.r.t. their generality. For example, type $\sigma_1 = \forall x.(y \to x) \to x$ is more general than type $\sigma_2 = \forall x.(x \to x) \to x$, because type σ_1 can be transformed to type σ_2 with substitution $[x/y]$. Type $\sigma_0 = \forall x.z \to x$ is even more general than σ_1 and σ_2. With respect to generality, type schemes can be ordered from the most specific type to the most general type with relation $<$ (more general), denoted as $\sigma_2 < \sigma_1 < \sigma_0$.

Figure 5 presents the implementation of type schemes in the Anemone compiler, realised with classes *Forall* and *Type*. Polymorphic type $\forall a.a$ would be implemented as `Forall("a", TypeApp("a"))`.

Hindley-Milner type inference is performed with six inference rules:

$$\mathit{TAUT} : \frac{}{A \vdash e : \sigma}\ (x : \sigma \in A) \tag{R1}$$

$$\mathit{INST} : \frac{A \vdash e : \sigma}{A \vdash e : \sigma'}\ (\sigma > \sigma') \tag{R2}$$

$$\mathit{GEN} : \frac{A \vdash e : \sigma}{A \vdash e : \forall \alpha \sigma}\ (\alpha \text{ is not free in } A) \tag{R3}$$

$$\mathit{COMB} : \frac{A \vdash e : \tau' \to \tau \quad A \vdash e' : \tau'}{A \vdash (ee') : \tau} \tag{R4}$$

$$\mathit{ABS} : \frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash (\lambda x.e) : \tau' \to \tau} \tag{R5}$$

$$\mathit{LET} : \frac{A \vdash e : \sigma \quad A_x \cup \{x : \sigma\} \vdash e' : \tau}{A \vdash (\text{let } x = e \text{ in } e') : \tau} \tag{R6}$$

Rule (R1) is a tautology which attributes type to an expression. Rule (R2) defines instantiating of a more specific type scheme. Rule (R3) enables type generalization w.r.t. a generic type variable. Rules (R4) and (R5) determine the type of function application and the type of function definition, respectively. Rule (R6) enables the typing of expressions wherever an introduced variable has been used.

According to the above rules, the only available types are simple types ι and function types, constructed with operator $\rightarrow$. In addition to simple types and function types, Anemone offers complex types (introduced with a type constructor, i.e., a function returning a new type). For example, a type representing a pair of values `Pair`: $a \times b$ can be defined with the following type constructors:

1) `PairConstructor`: $\forall ab.\ a \rightarrow b \rightarrow a \times b$
2) `GetFirst`: $\forall ab.\ a \times b \rightarrow a$
3) `GetSecond`: $\forall ab.\ a \times b \rightarrow b$

This approach is used in the compiler of Anemone. It introduces appropriate type constructors for each data structure defined in Anemone.

```
sealed trait TypeScheme

case class Forall(name: String,
                  typeScheme: TypeScheme)
     extends TypeScheme

case class Type(typeTerm: TypeTerm)
     extends TypeScheme
```

Fig. 5: Implementation of type schemes in the compiler of *Anemone*

C. Algorithm W

Algorithm *W* [8] implements Hindley-Milner type inference and is the basis for type reconstruction performed by the Anemone compiler. Algorithm *W* works according to formula $W(A, e) = (S, \tau)$, where A is a set of variables with known types and e is an expression, the type of which is reconstructed. The result of the algorithm is a pair (S, τ), where S is the principal unifier needed to find the type of expression e, and τ is the reconstructed type of expression e.

Within the algorithm *W* a closure $\overline{A}(\tau)$ of type τ at assumptions *A* is defined as

$$\overline{A}(\tau) = \forall \alpha_1, \alpha_2, .., \alpha_n.\ \tau\ ,$$

where $\alpha_1, \alpha_2, .., \alpha_n$ are free type variables in τ and do not occur in A.

Algorithm 1 presents the original form of algorithm *W*, which consists of four rules. Each rule corresponds to one of four production rules of grammar in Fig. 4a and inference rules (R1)–(R6).

- The first rule of algorithm *W* corresponds to inference rules *INST* preceded by rule *TAUT*;
- the second rule of algoritm *W* corresponds to inference rule *COMB*;
- the third rule corresponds to rule *ABS*;
- the fourth rule corresponds to rules *LET* and *GEN*.

```
1) if $e$ is identifier $x$ and hypothesis about its type
   belongs to the set of known hypotheses
   $x\colon \forall \delta_1, \delta_2, .., \delta_n.\ \tau'$ then
       $S = [\ ] \quad \tau = [\gamma_i/\delta_i]\tau'$ ,
   where $\gamma_i$ is a new type variable.
2) if $e$ is function application $e_1 e_2$ and:
       $W(A, e_1) = (S_1, \tau_1)$
       $W(S_1 A, e_2) = (S_2, \tau_2)$
       $\mathit{Unify}(S_2\tau_1, \tau_2 \rightarrow \gamma) = V$ ,
   where $\gamma$ is a new type variable then
       $S = V S_2 S_1 \quad \tau = V\gamma$ .
3) if $e$ is function abstraction $\lambda x.e_1$ and:
       $W(A_x \cup \{x\colon \gamma\}, e_1) = (S_1, \tau_1)$ ,
   where $\gamma$ is a new type variable then
       $S = S_1 \quad \tau = S_1\gamma \rightarrow \tau_1$ .
4) if $e$ is expression let $x = e_1$ in $e_2$ and:
       $W(A, e_1) = (S_1, \tau_1)$
       $W(S_1 A_x \cup \{x\colon \overline{S_1 A}(\tau_1)\}, e_2) = (S_2, \tau_2)$ ,
   then
       $S = S_2 S_1 \quad \tau = \tau_2$ .
```

Algorithm 1: Algorithm *W*

D. Implementation of algorithm W

Figure 4c presents the grammar of the language of expressions used in the Anemone compiler. In comparison to the grammar of L_e (Fig. 4a), there are the following changes:

- replacing definitions and calls of a unary function with definitions and calls of n-ary function ($n > 0$)
- introduction of *fix* operator, which enables definitions of recursive functions
- introduction of *noarg* operator, representing definitions of functions with no arguments

As Anemone enables n-ary functions, appropriate constructs supporting n-ary functions have been introduced. Modelling n-ary functions with operators acting on unary functions would imply that a call of a binary function with only one argument provided is correct as (from the point of view of implementation of algorithm *W*) a function would always take only one parameter. Such an implementation of algorithm *W* was considered; however, it would impose implementation of partial parametrisation (currying) for each function in Anemone. While interesting, this approach would require significant modifications in the entire compiler. The solution

1) **if** e is identifier x **then** proceed according to the steps in Algortihm 1.
2) **if** e is of the form $\lambda(x_1, x_2, .., x_n).e$ **and**:

   $$W(A_{x_1x_2..x_n} \cup \{x_1\colon \gamma_1, x_2 : \gamma_2, .., x_n\colon \gamma_n\}, e) = (S_1, \tau_1) ,$$

   where $\gamma_1, \gamma_2, .., \gamma_n$ are new type variables, **then**:

   $$S = S_1 \quad \tau = (S_1\gamma_1, S_1\gamma_2, .., S_1\gamma_n) \to \tau_1 \ .$$
3) **if** e is a operator of zero-argument function, $noarg(e_1)$, **and**:

   $$W(A, e) = (S_1, \tau_1) \ ,$$

   then:

   $$S = S_1 \quad \tau = unit \to \tau_1 \ .$$
4) **if** e is a multiargument function call, $e_1(e_2, e_3, .., e_{n+1})$, **and**:

   $$W(A, e_1) = (S_1, \tau_1) \ ,$$

   $$\underset{2\le i\le n+1}{\forall} : \quad W(S_1A, e_i) = (S_i, \tau_i) \ ,$$

   $$\mathit{Unify}(S_{n+1}S_n..S_2\tau_1, (\tau_2, \tau_3, .., \tau_{n+1}) \to \gamma) = V \ ,$$

   where γ is a new type variable, **then**:

   $$S = VS_{n+1}S_n..S_1 \quad \tau = V\gamma \ .$$
5) **if** e is expression `let` $x = e_1$ `in` e_2 **then** proceed according to the definition in Algorithm 1.
6) **if** e is a fixed point expression $\mathit{fix}(x, e_1)$ **and**:

   $$W(A_x \cup \{x : \gamma\}, e_1) = (S_1, \tau_1) \ ,$$

   $$\mathit{Unify}(S_1\gamma, \tau_1) = V \ ,$$

   where γ is a new type variable, **then**:

   $$S = VS_1 \quad \tau = VS_1\gamma \ .$$

Algorithm 2: Extended version of algorithm *W* applied in *Anemone*

chosen for implementation introduces constructs which support representations of n-ary functions at the level of the algorithm *W* as it limits necessary reorganisations to the module of type reconstruction.

Algorithm 2 presents the extended version of algorithm *W* applied in Anemone, which takes into account new constructs and changes in grammar of L_e.

E. Implementation of let-polymorphism

Anemone is equipped with a kind of parametric polymorphism known as let-polymorphism. Its name stems from the syntactic construct present in *ML*, where this kind of polymorphism has been introduced. The *let* construct, also present in the definition of algorithm *W*, introduces generically typed expressions.

Anemone does not have keyword *let* which serves to introduce polymorphic functions, as is done in *ML*. Moreover, *AST* representing Anemone code is more extensive than the simple language of expressions, L_e (Fig. 4a), on which algorithm *W* is based. Also, the grammar of type expressions, presented in Fig. 4b, is much simpler than the set of types possible to express in Anemone, due to the possibility to construct types of structures.

When implementing algorithm *W* for Anemone, it was possible to choose between two approaches. The first approach extends the definition of algorithm *W* in Algorithm 1, in order that the implementation also comprises constructs expressed by particular *AST* nodes of Anemone. The second approach transforms *AST* to a simpler form corresponding to the language of expressions L_e.

In the Anemone compiler, the second approach has been chosen. In the phase of type reconstruction, *AST* is transformed to a simpler form used in the implementation of algorithm *W*, which we shall call *WAST*. The implemented version of algorithm *W* uses a slightly richer grammar of expressions, defined in Fig. 4c.

Usually, a structure containing many *WAST* nodes is created from each *AST* node, and extension of algorithm *W* for the purpose of Anemone facilitates this transformation with moderate complication of implementation of algorithm *W*.

In particular, the algorithm of type reconstruction in the Anemone compiler proceeds as follows:

1) Nodes introducing new types are separated from nodes which are a subject to type reconstruction. Nodes introducing new types are declarations of new data types and declarations of external functions.
2) Each external function and its type is added to the set of known assumptions about types.
3) For each declaration introducing a new data type, the new type and its name are added to the set of known assumptions about types.
4) The set of known assumptions about types is complemented with types corresponding to built-in operators and primitive data types,
5) The set of known types is converted to the form in Fig. 4b, i.e. type schemes, simple types and type constructors – similarly to type `Pair` in Sect. IV-B.
6) The remaining *AST* nodes are converted to subtrees of *WAST*. It is recorded which *AST* node corresponds to a given subtree of *WAST*.
7) Algorithm *W* is executed on *WAST* representation of a program, with assumptions about types gathered from structure declarations, external functions, and built-in operators and types.
8) The obtained principal unifier is applied to all *WAST* nodes in order to obtain the most general type for each node.
9) For each *WAST* node corresponding to *AST* node, its type is mapped to type representation used in *AST*.

As a result of using dedicated representation of *WAST* code during type reconstruction, the entire process of type reconstruction is independent from remaining modules of the implemented compiler. Moreover, the unification algorithm and algorithm *W*, which are crucial for the process of type reconstruction, operate on relatively simple, abstract data types dedicated to this task. Above features guarantee strong resistance of the entire module to changes implemented in other modules of the compiler and enable its seamless extension in the future.

F. Code generation for polymorphic functions

Polymorphic functions must be compatible at the binary level. For example, mapping function *map* takes as its arguments a list of elements and a function to be executed on each element of the list. A parameter responsible for this function receives type $\forall x.\ x \to y$, where x and y are type variables. Without knowing concrete values of variables x and y, their representation for the target architecture should be chosen. Let be given a memory pointer. Let us define *minus* function taking a value of primitive type *double* and returning its negation. Type of function *minus* will be instantiated as *double* $\to$ *double*. Assembly code of this function must be independent from information, whether function argument is of floating-point type. During a function call in a given architecture, an argument of floating-point type may be passed differently (e.g., in a different register) than a value of a pointer. If we pass to function *map* function *minus*, implementation of *map* will call the passed function as if it would take a parameter being a pointer, whilst function *minus* expects a floating-point type. Such an incompatibility can easily lead to errors in a compiled program.

```
f:     # type::('a, 'b) -> unit
...
movq    %rdi, (%rsp)
movq    %rsi, 8(%rsp)
movq    %rdx, 16(%rsp)
...

g:     # type::(double, 'b) -> unit
    ...
movq    %rdi, (%rsp)
vmovsd  %xmm0, 8(%rsp)
movq    %rsi, 16(%rsp)
...
```

Fig. 6: Assembly code *x86_64* generated for functions *f* and *g*

Functions *f* and *g* have similar signatures (given after # sign). Consistently with the implemented in Anemone type system, usage of function *g* can be replaced with usage of function *f* because generic type variable *'a* can be unified with type *double*. If generated assembly code treats first argument of function *g* as a floating-point number, then value of this argument is obtained from register *xmm0*. In contrast, when first argument of function *f* is treated as a pointer, its value is obtained from register *rsi*.

Figure 6 presents different assembly codes of two functions, taking a pointer and a floating-point number, respectively, as a parameter. To avoid the problem of binary incompatibility between functions, Anemone compiles all the function parameters as pointers. This solution is known as boxed representation [11]. The potential drawback of this solution is reduced efficiency of generated code; however, it guarantees correctness of programs with polymorphic functions. An alternative approach for compiling polymorphism uses intentional type analysis [12].

V. Conclusions

The type system is one of the strongest points of the Anemone language. Full type inference disposes the programmer from manual defining of type labels. Moreover, let-polymorphism enables function definitions with respect to generic parameters of type. In this way, a separate function implementation for each type becomes redundant and code conciseness is promoted.

Support for algebraic data types and pattern matching enables rich data structures, and easy implementation of lists, trees or other data structures. The implemented type system guarantees static correctness of a program while preserving code conciseness and expressiveness.

Appendix

Figure 7 presents basic constructs of Anemone. The example defines functions `fib` and `factorial` of type `double -> double` which use conditional expressions `if-else`. Function `factorial` defines nested function `helper` inside its body.

```
type:: (double) -> double
fun fib(n) {
    if(n == 0) {
        1
    } else {
        if(n == 1) {
            1
        } else {
            fib(n-1) + fib(n-2)
        }
    }
}

type:: (double) -> double
fun factorial(m) {
    type:: (double, double) -> double
    fun helper(ax, n) {
        if(n == 0){
            ax
        } else {
            var n2 = n - 1 in {
                helper(ax*n, n2)
            }
        }
    }
    helper(1,m)
}
```

Fig. 7: Basic constructs of the Anemone language

Functions in Anemone are first-class citizens, as they can be: assigned to a variable (Fig. 8), passed as function arguments (Fig. 9), returned from a function (Fig. 10), and handled as a closure with a non-empty environment (Fig. 11).

```
fun foo(){
        fun bar(){
                1
        }
        bar
}
```

Fig. 10: Returning function from a function

```
fun buildAdder(n){
        fun adder(x){
                x + n
        }
        adder
}
```

Fig. 11: Function being a closure with non-emtpy environment

```
data BoolList = BoolCons {
    value :: boolean,
    next :: BoolList
} | BoolNil { }
```

Fig. 12: Definition of new multi-variant data type

```
data List 'a = Cons 'a {
    value :: 'a,
    next :: List 'a
} | CNil { }
```

Fig. 13: Definition of polymorphic data type. Variable `a` is a type variable

```
fun foo(l) {
        match l {
                | Cons(v, n) => {
                        bar(v, n)
                }
                | cnil :: CNil => {
                        baz()
                }
        }
}
```

Fig. 14: Usage of pattern matching mechanism

```
fun bar(x){
        var f = foo in {
                f(x)
        }
}
```

Fig. 8: Assignment of a function `foo` to a variable

```
fun apply(f, x){
        f(x)
}
```

Fig. 9: Passing a function as a parameter

Anemone defines three primitive types: `boolean`, `double` and `string`. Examples in Figs. 12 and 13 define a multi-variant data type `BoolList` and polymorphic data type `List`.

Anemone also offers pattern matching, as shown in Fig. 14. There are two kinds of patterns. Pattern `| Cons(v, n) =>` is similar to deconstructive assignment, pattern `| cnil :: CNil =>` is similar to declaration of a variant field.

Concurrency in Anemone is implemented with actors [1], [2]. Actor model of *Anemone* has the following features:

- asynchronous communication with message passing, which is clearer and less error-prone than thread model and makes manual synchronization redundant
- actors modelled as functions (similarly to Erlang) and actor function called for each new message (similarly to the Akka library).
- possibility of creating many actors in one thread
- possibility of dynamic creation of new actors
- identification of received messages through pattern matching
- message passing model implemented with shared memory
- complete integration with the garbage collector

An example of creating an actor system with two actors communicating with each other via messages is given in Fig. 15.

Acknowledgments

The research presented in this paper was supported by the funds of Polish Ministry of Education and Science assigned to AGH University of Science and Technology.

References

[1] P. Batko and M. Kuta, “Actor Model of a New Functional Language - Anemone,” in *Proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics, PPAM 2017*, ser. Lecture Notes in Computer Science, vol. 10778, 2018. doi: 10.1007/978-3-319-78054-2_20 pp. 213–223.

[2] ——, “Actor model of Anemone functional language,” *Journal of Supercomputing*, vol. 74, no. 4, pp. 1485–1496, 2018. doi: 10.1007/s11227-017-2233-1

[3] A. Church, “A Formulation of the Simple Theory of Types,” *Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940. doi: 10.2307/2266170

[4] J. McKinna, “Why Dependent Types Matter,” *SIGPLAN Notes*, vol. 41, no. 1, pp. 1–1, 2006. doi: 10.1145/1111320.1111038

[5] A. Bove, P. Dybjer, and U. Norell, “A Brief Overview of Agda — A Functional Language with Dependent Types,” in *22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, 2009. doi: 10.1007/978-3-642-03359-9_6 pp. 73–78.

[6] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002. ISBN 978-0-262-16209-8

[7] A. J. Kfoury and J. B. Wells, “Principality and Decidable Type Inference for Finite-Rank Intersection Types.” in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL’99*, 1999. doi: 10.1145/292540.292556 pp. 161–174.

[8] L. Damas and R. Milner, “Principal Type-schemes for Functional Programs,” in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL’82*, 1982. doi: 10.1145/582153.582176 pp. 207–212.

[9] J. A. Robinson, “A Machine-Oriented Logic Based on the Resolution Principle,” *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, 1965. doi: 10.1145/321250.321253

[10] R. Hindley, “The principal type-scheme of an object in combinatory logic,” *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969. doi: 10.2307/1995158

[11] M. Hicks, “Types and Intermediate Representations,” Department of Computer and Information Science, University of Pennsylvania, Technical Report MS-CIS-98-05, 1998.

```
fun ping(state, msg) {
  var otherActorId = state in {
    match msg {
      | s :: String => {
          printStr(s)
          sendMsg(otherActorId, "fromPing")
          nap(1)
          state
      }
      | otherActorId :: ActorId => {
          sendMsg(otherActorId, "fromPing - first")
          otherActorId
} } } }

fun pong(state, msg) {
  var otherActorId = state in {
    match msg {
      | s :: String => {
          printStr(s)
          sendMsg(otherActorId, "fromPong")
          nap(1)
          state
} } } }

fun main_fun() {
  createActorSystem(2)
  var pingActorRef = createActor(ping, 0),
      pongActorRef = createActor(pong, pingActorRef) in {
        sendFromOutside(pingActorRef, pongActorRef)
} }
```

Fig. 15: Creating and starting an actor system

[12] R. Harper and G. Morrisett, "Compiling polymorphism using intensional type analysis," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, 1995. doi: 10.1145/199448.199475 pp. 130–141.