Acta Informatica Pragensia
1(1), 2012, 1–21, DOI: 10.18267/j.aip.1
Online: aip.vse.cz

Section:
Peer-reviewed papers

Will we think in programming languages?

Jiří Fišer[1]

[1] Department of Informatics, Faculty of Science,
Jan Evangelista Purkyne University in Usti nad Labem
Ceske mladeze 8, Usti nad Labem 400 96, Czech Republic

jf@jf.cz

Abstract: Modern science commonly uses computer modelling. Thousands of scientific model are daily transformed to computers programs and tested. The transformation must overcome the gap between abstract human formal notation and low level semantics of contemporary programming languages. The simultaneous knowledge of specific scientific models and programming languages is an unpleasant necessity for a significant proportion of scientists and practitioners (engineers, economist, etc.). But a solution exists — accommodation of programming languages to mental models of their users. The article discuss one partial solution — implementation of domain specific languages in the heart of existing universal languages by mechanisms of metaprogramming. This mechanism overcomes limitations of classical programming languages and complexity of creation new languages from scratch. However, the support of metaprogramming in contemporary languages is limited to isolated and peripheral constructs in very few languages. These constructs are demonstrated by simplified but real examples (metaobject system of Python, monads in F# and macro-based metaprogramming of Boo language) together with discussion of their advantages and disadvantages. The discussion of examples is aimed to finding requirements for new languages and their implementation in a original (parent) language.

Keywords: Programming languages, Domain-specific languages, Metaprogramming, Python, F#, Boo

1 INTRODUCTION

The programming languages have been existing among us over sixty years. However the basic mechanisms of its application has not been changed anyway. Some person (scientist, businessman, office worker, etc.) has an idea and he (or she) is capable transform it into a formal model (the formalisms needn't be based on mathematical notation). The practitioner (= person with ability of producing some formal models) wants to applicate this model as IT solution or check its validity by computer simulation. Therefore practitioner contacts other person of special kind (or team of these persons). This person has ability to communicate with computers by means of programming languages and ergo he (or unfortunately very rarely she) is called *programmer* (at lest by himself). The programmer transforms the model into source code of his (or her) favourite programming language and compiler transform the source code into executable computer program (see Figure 1, section "current model") .

This (very simplified) chain of responsibilities has a lot of disadvantages. Firstly, it is error-prone because the practitioner doesn't understand programming languages and programmer doesn't cope with the models and especially with their context and background. Secondly, it is very expansive because the superior programmers are very rare (and therefore must be overpaid) and for communication with normal sort of programmers the practitioner needs assistance of a mediator, which interprets or translates the model to limited language of programmer (this mediator is called software designer of analyst).

There is only one prefect solution of these problems — *to make programmer from all persons*, which are or potentially will be originators of formal models. This ideal goal is reachable by two ways:

1. *by teaching of universal programming languages in high schools and universities*. This solution has been very popular but unfortunately only small part of practitioners are fully familiar with universal programming lan guages. The main reason is obvious — the standard mechanism of programming languages are very distant from formalism of other science disciplines and contemporary programming and particular disciplines are both very complex systems and there is not possibility to shrink both to one university curriculum. Furthermore even genii which manage both programming and particular discipline must intricately transform their models to program codes.
2. *by accommodation of programming languages to mental models of practitioners*. The fully specialized programming language have a potential to replace practitioner's formal language in his brain, because it offers both alternative representation of formalism and medium for (inter-human) communication. Moreover, this language could be a bridge between practitioner and computer, simultaneous natural for both human being and computer (after compilation), see Figure 1 section "ideal model". This final goal — *thinking in programming language* — is very distant in time, but some modern programming languages suggest a way.

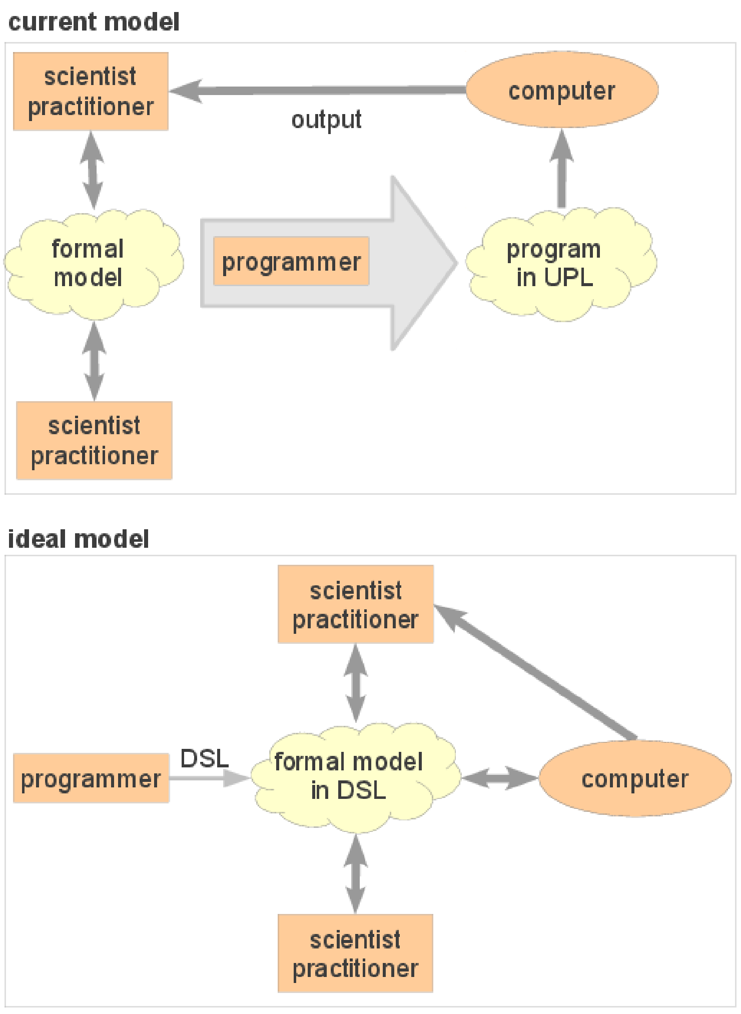

Fig. 1. Current and ideal model of cooperation with a computer system.

2 HISTORICAL BACKGROUND

The first step on this way was made in computer prehistory: genesis of high level programming languages (late 1950's). The high programming languages have been designed for humans not for computers, but their mechanisms and tools are heavily based on contemporary computer models.

The three most prominent primordial high level programming languages exemplify limitations of this stage of development.

Fortran (1957) — programming language designated for scientific computing. The syntax is close to basic algebraic notation, but there are constructs, which reflects hardware details (e.g. explicit memory management). The data representation was limited to numbers and their arrays (complex data structures are not directly representable). The imperative paradigm of Fortran overuses assignment semantics (the most important Fortran statement is assignment) and (poorly implemented) loops. Fortran forces its user to thinking in its very strict but also restrict way (from the point of view of untouched practitioner)

Cobol (1959) — programing language designed for business, finance and administrative systems. Cobol has very verbose syntax which mimics English (but the syntax and semantics model is totally different). Cobol had more wider type systems, but this system was closed as well as operational semantics (control structures). The syntax is not structured in the way of mainstream languages. Edsgar Dijkstra remarked that “The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence” [2]. The interpretation of this remark is ambiguous. On one hand, it illustrates the negative result of tendency of accommodation of practitioners to inadequate programming language, on the other hand the remark may reflect false superiority of one paradigm (probably pure mathematical based structural programming, which was preferred in the time of remarks).

Lisp (1958) — programing languages used for artificial intelligence research. The *Lisp* pioneered some very advanced mechanisms as tree-like data structures and automatic memory management. But its syntax with parenthesized Polish notation is very different from typical human-oriented formalism (which is mostly more terse). This exotic notation degrades the main contribution of Lisp, the macro system, which allows modification of basic syntax of language and even creation of a new (more accommodated) language system (unfortunately with Lisp parenthesized syntax on surface).

The main deficiency of primordial languages is very limited extensibility (only subroutines) and simplicity of their internal data and operational models (*Lisp* is partial exception). Moreover all these languages have been designed by mathematician (Fortran by *John Backus*, Lisp by *Steve Russell* and Cobol by *Grace Hopper*) and are based on mathematical works of founder of computer science. Even at seventies Dijkstra wrote [2]: “Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians”. This origin has to leave a mark on programming languages.

The sixties brought two trends into the field of practical programming languages: large universal languages (PL/1, Algol 68) and small specialized languages for simulations.

The large universal languages have been designed for all purposes by merging of constructs of older languages. The resulted mastodons lack orthogonality and were superfluously complex even for professional programmers. Therefore in the seventies, they have been replaced by much simpler universal languages (*Pascal* and *C*), which have been very popular among coders, but their usability for practitioners is questionable. The reason is similar to primordial languages — very limited extensibility and immanent simplicity (for example *C* directly support only numerical and pointer arithmetics).

The research in the field of specialized simulation language in the 1960’s was mostly ignored by mainstream computer scientists and programmers. But the specialized language had some principal advantages:

1. some languages have been designed by practitioners of concrete science discipline (economists, engineers) and therefore they are modeled on real formalism
2. the huge quantity of specialized simulation language and its mutual influences make possible evolution by “natural selection”.

But these languages have also the dark side:

1. specialized simulation languages have very simple and in many cases inconsistent syntax. The reason is obvious: the design of languages from scratch and implementation of a decent compiler is very challenging task (and was even more challenging in the 1960's).
2. the stability and long term support were very problematic in the world of thousands languages, which are revised almost every year (and by teams of one person)

However, from this gray and shadow zone started the *second revolution of programming languages* — data abstraction based on object oriented methodology (Simula 1968, [1]). This revolution was only partial because it founded new data oriented paradigm on the base of the classical universal imperative language (Algol 60), and for at least decade was confined only to periphery (the most influential books of 1970's programmers do not mention OOP anyway, e.g. *Wirth's Algoritms + Data structures = Programs*, [18]).

3 PRESENT SOLUTIONS

Presently the object oriented programing is leading methodology of programming world and OOP is almost essential part of any programming language (OOP extensions exist even for *Fortran* and *Cobol*). Unfortunately, the majority of languages are only would-be object oriented languages which do not exploits the potential of OOP abstraction for the process of accommodation of programming languages to mental models of practitioners. I do not mean only residuals of pre-OOP data methodologies in programming languages (e.g. classical type system in C++) but also the archaic approaches to learning programming for practitioners (and often also for programmers) [13].

The OOP revolution in data abstraction has been accompanied by positive movement in type system mechanism, from tools which facilitate the use of complex data structures in statically typed programming languages (generics and automatic type inference), to sophisticate metaobject protocols of dynamic typed languages.

The *metaobject protocols* (MOP) can alter basic data semantics by extending or changing of internal structures of interpreter, which are exposes by a public interface. The metaobject protocol is relatively easy implementable in *homoiconic languages*. The language is homoiconic if the data representation of program code is also a primitive type of language itself (e.g. *Lisp*). Nowadays some kind of metaobject protocol is implemented in many popular object-oriented scripting languages (*Python*, *JavaScript* [6], *Ruby* [9]).

Application of complex classes together with the specialized data semantics based on *metaobjects protocols* [4] launched a new era for *domain specific languages* (standard abbreviation is DSL). These languages are descendants of simulation languages of the sixties and also modeling declarative languages.

The contemporary domain-specific languages can been characterized by:

1. (extensive) declarative parts for easy definition of static parts of models. Some DSLs are fully declarative (e.g. *graphviz* or XML languages) but these are not programming languages in *stricto sensu*.
2. a comprehensive OOP library of classes, which is targeted to concrete problem domain (and reused in the context of DSL). This library is, in typical case, closed code (i.e. it isn't expandable inside DSL)
3. an auxiliary language for manipulation with object and extension of operational part of declarative code. The most widely used language for this purpose is *JavaScript* (other possibilities: *Lua*, *Scheme*). In all cases, this language is a simple imperative language with classical control constructs and ordinary interaction with OOP objects.

For simplicity: *typical DSL = declarative definitions + fixed OOP library + simple auxiliary imperative language*. This approach has some pros (low cost, reusing of existing solutions) but also unquestionable cons:

1. very limited scalability: the DSL itself doesn't support creation of domain specific classes or even new language constructs (both declarative and imperative)
2. syntactic and semantic gap between declarative and imperative code. The imperative code play the role of glue (or vice versa), but there is no possibility freely intermix declarative and imperative code
3. the semantic density of DSL code varies. The declarative code is generally denser than imperative one.

Metaobjects protocol of universal programming languages make possible completely different kind of domain specific languages — DSL created inside universal language. But MOP offers only one dimension of freedom — freedom of data representation and direct data manipulation (superstructure of class interface of classical DSLs). The second dimension — the freedom of implement new operational syntax and semantics needs an easy tool for metaprogramming i.e. programming of program constructs. The advanced metaprogramming will make possible revolutionary alternation of basic syntax of a language including replacement of surface syntax. For completeness, the third dimension is support of extensible declarative syntax interleaving procedural code (not discussed here but for example of real solution see [8]).

The domain specific languages embedded inside universal languages have been implemented and thoroughly discussed in *Lisp* since 1960's. However the Lisp language and its dialects uses unfamiliar syntax (unfamiliar for main stream programmers and likely practitioners). Current research in the field of metaprogramming make possible support of metaprogramming in more convenient language (e.g. in *Haskell* with esoteric functional semantics but modern and reasonable syntax[14]) and even in almost usual procedural language as *Converge* (with syntax similar to *Python,* but semantics based on Icon). The implementation of metaprogramming language in Converge reuses some constructs based in Lisp semantic macros and quasi-quotations, but also innovates specialized DSL constructs[16].

These researches have only little direct impact in the field of main stream programming because they are not supported in standard language and common compilers (case of *Haskell* metaprogramming) or the language is peripheral scholar project (case of *Convert*). However outputs of fifty years of research is starting to manifest in more popular programming languages.

4 METAPROGRAMMING AND METAOBJECTS IN MAIN-STREAM LANGUAGES

The mainstream languages which offer all these freedoms and make possible and easy creation of fully-fledged DSLs aren't yet available but some isolated features are already implemented and exploited in several present-days languages.

I have chosen three typical use cases:

- **Python** (www.python.org) and its metaobject protocol and decorators
- **F#** (http://research.microsoft.com/en-us/projects/fsharp/) and simpler OOP way to functional monads
- **Boo** (http://boo.codehaus.org) and its high level metaprogramming

4.1 PYTHON AND FLEXIBLE METAOBJECTS PROTOCOL

The Python language is widely used universal language with a relatively long history (1991). Python has been language with a multiparadigm support and relatively uncomplicated and sparse syntax. However Python was language which support only proven language mechanisms and standard solutions. New metaobject protocol which has been implemented since Python 2.2 (2001) and clarified in Python 3 opens the door for more ambitious syntax mechanisms.

The Python have two interfaces to MOP: *metaclasses* (low level tool) and *decorators* (high level and more explicit) [15].

See following short fragment of Python code:

```
class Image1D(cp_object):
    "format: image = {.img}"
    def Image1D(self):
        self.img = []

    @noise(1.0)
    def add(self, x):
        self.img.append(x)

img = Image1D()
print(img.add(10).add(10).add(10))
#example  of  output:  image  =  [7.499177565043477,  9.795160236072121,
11.503326240163346]
```

The surface syntax of the fragment is clearly pythonic (indentation, separators, keywords, explicit *self*). But more detailed view exposes some discrepancies:

- the constructor is identified by class name as in C++ or Java (in Python the constructor has name *__init__*)

- the method *add* doesn't return any value (in Python these methods formally return None *value*), but chain of method call in print function assumes the returning of self object
- the class doesn't define *__str__* special method (analogy of *toString* of Java) but printed value is usable and formated (usual output without *__str__* contains only class name). Moreover, the output format is magically inferred from document string (free string literal after class header)
- the final (printed) status of object contains list of three random values, but value are added without explicit randomization (see body of method *add*). This randomization is possibly related to declarative notation *@noise(1.0)* before *noise* method header.

The reason is obvious, the fragment is not implemented in Python but in slightly modified sublanguage (i.e. language defined inside Python). This language is modified in two directions:

- the class definition is simplified (by inclusion of ideas from other languages). The target group of these changes are programmers, which dislike pythonic special names (with plenty of underscores) and massively use chains of method calls. The impact for practitioners is undirect (only as general simplification).
- the processing of values with inherent noise is simplified. This extension is more targeted for practitioners.

The simplification of class definitions is implemented by metaclass — factory object for classes (classes in Python are objects)

```
def wrapper_factory(f): #create wrapper as closure
    def wrapper(*args, **kwargs):
        rv = f(*args,**kwargs)
        if rv is None:
            rv = args[0] #=self
        return rv
    return wrapper

class cp_meta(type):
    "metaclass"
    def __new__(cls, className, baseClasses, dict):
        #creation of __str__ method from doc-string
        if "__doc__" in dict:
            m = re.match(r"format:\s(.*)", dict["__doc__"])
            if m:
                format=m.group(1)
                dict["__str__"] = lambda self : format.format(self) #1
        if className in dict: #new name for constructor
            dict["__init__"] = dict[className]
            del dict[className]
        for id, val in dict.items(): #wrappers for methods without return value
            if callable(val) and not id.startswith("__"):
                dict[id] = wrapper_factory(val)
        return super().__new__(cls, className, baseClasses, dict)

class cp_object(metaclass=cp_meta): #formal base class
    pass #empty body of class
```

The implementation is slightly intricate, but it is relatively comprehensible for professional coders. Users of derived language (i.e. practitioners) do not come into contact with a supporting code.

The code exposes some general (positive) characteristics of derived language implementation:

hiding of implementation mechanism

The user of derived language should be strictly isolated from complex mechanisms, which have been used for implementation of this language. In our code the inheritance is used for isolating. The user classes must be derived from *cp_object* class, because all directly or indirectly derived classes have to share the same metaclass in order to exhibit behavior of classes of derived language (see UML class diagram in Figure 2).

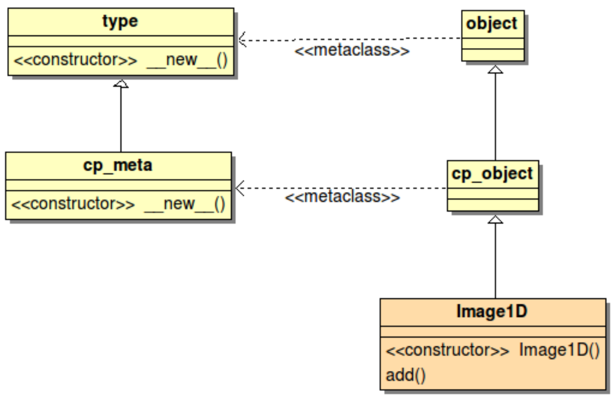
type
<<constructor>> __new__()
<<metaclass>>
object
cp_meta
<<constructor>> __new__()
<<metaclass>>
cp_object
Image1D
<<constructor>> Image1D()
add()

Fig. 2. UML class diagram of classes and metaclasses.

The inheritance is standard and widely used OOP mechanism, which is available also in derived language.

correct coexistence with parent language

The derived language exists int the context of parent language and therefore the coexistence of both languages must be taken into account. The correct coexistence requires that any syntactical construct of original language, which isn't intentionally changed, is either

A) unreachable from derived language (= they are syntactically incorrect or at least causing runtime exception)

or

B) maintaining original syntax semantics

Our prototypical implementation breaks this requirement because:

- the classical constructor with name *__init__* is definable but it isn't callable, if there exists new constructor with class name (simplest solution: exception thrown in code of metaclass, if the *__init__* method exists)

- the special method *__str__* is not used, if a format string exists in documentation string (simplest solution: if explicit *__str__* method exists, the format string would be skipped)
- if the method explicitly returns *None* (e.g. return value is unavailable), the code of wrapper in metaclass will change it to self-object. Ideal solution doesn't exist because in Python the implicit *None* (returned form method-procedure without return-statement) is indistinguishable from explicitly returned *None*. The problem is broader because the majority of programming language do not distinguish undefined and unavailable values (rare exception is *JavaScript*)

efficiency of derived language

The efficiency of derived language should be similar to efficiency of its original language.

In our example the efficiency of constructors and formatter is almost identical with the efficiency of their counterparts (i.e. no additional memory is need and activation isn't visibly slower). The mechanism of implicit returning of self-object requires auxiliary wrapper function. This function require additional memory and the activation of all methods is slower (all methods are wrapped excluding only special methods). The wrapper adds one method call and one test of equality with *None* (which fortunately doesn't depend on magnitude of tested object) i.e. the total time complexity is still $O(1)$.

The alteration of class semantics based on metaclasses isn't explicitly expressed. This approach is reasonable only if the new semantics is universally usable in targeted domain. But some alteration are usable only in very limited contexts, e.g. for proper subsets of class or methods. From this point of view, not only one derived language is implemented, but suite of simple languages which coexist (and even cooperate) in one source code.

Python provides this approach (or foretaste of this approach) by system of decorators (for real example of decorator-based language see [17]). Decorators are applied to method and classes and they have ability to change semantics of these constructs. The decorators are superficially declarative notations (similar to Java annotations), but actually they are functions which transform one method to another one or a class to another class. The semantic strength of class decorators isn't much less than strength of metaclasses.

For brevity, our example utilizes only one simple method decorator. The decorator *noise* facilitates the processing of values with Gaussian noise. The each input numerical value of decorated method is automatically equipped with random Gaussian noise (analogy of inherent noise of input signals).

The implementation of decorator is straightforward (note the multiple use of function wrappers in the form of closures).

```
def noise(sigma): #sigma = standard deviation of noise
    def decorator(f): #closure applied to method
        def wrapper(*args): #wrapper of the method
            newArg = [
                arg + normalvariate(0.0,sigma)
                if isinstance(arg, numbers.Real) else arg for arg in args]
            return f(*newArg) #call of original method
        return wrapper
        return decorator
```

The random value with Gaussian distribution is added only to parameters with numerical values (i.e. object of class which falls under the abstract base class *numbers.Real*). This constraint prevents only trivial type errors, but more subtle faults (unwanted side effects) remains:

- modification of additional numerical parameters, which do not represent signal (e.g. simple counts, values of enumeration)
- multiple (repeated) modifications of one values (for example by nested calls of decorated methods inside another decorated method)

These problems reflect inadequateness of type system of bare Python, which support only two basic numerical types: *int* and *float*. In addition, these types have similar semantics and are interchangeable in many contexts (they are primarily defined by different memory representation and type of CPU processing). Fortunately, the creation of new specialized types are fully supported by Python type system (in the form of user defined classes) and new types are almost indistinguishable from built-in types (excluding literals which are available only for some built-in types).

In our case, two new numerical types (classes) are necessary: one for signal values without noise and other for values with random noise. These new types aren't ad hoc workarounds but integral part of model of new (customized) language i.e. the type distinction is natural and must be applied consistently in entire language (unfortunately slightly complicating the implementation, including our decorator).

The ability of Python to change language constructs (and construct new derived languages) is limited to data representation and data protocols. The universal operational syntax (assignments and other imperative statements) is unchangeable.

4.2 F# AND MONADS

The most perspective construct for enhancement of general operational semantics was originated in 1990's in the area of advanced functional programming. This construct is known as *monad*.

The monads are implemented or at least implementable in any language which supports functions as first class values (functional languages and some modern OOP languages e.g. *Python*, *Perl*, *Scala*, *Ruby*) but the practical language customization requires some syntactic sugar which is available only in a few languages (Haskell, F#)

The monads have been supported in Haskell since 1996 (Haskell 1.3) and they have played very important role in this language forming imperative sublanguage which is necessary for input and output routines (I/O monads) [7].

F# is relatively new functional language (2005) but it is strongly influenced by the older *Caml* language (1985) and its dialect *OCaml* (1996) (from the point of view of tradition in the field of functional programming, the F# is dialect of *Caml* too).

The *Caml* language doesn't provide syntactic sugar for monads because the monads are not essential part of language (the imperative construct are tolerated in this language). In the F# (2.0+) the support of monads is more advanced and monads are used for implementation of some language mechanisms (e.g. asynchronous workflows). I addition, the usability of monads is enhanced by utilization of more general syntactic construct of *computation expressions* [12, 10] (monads are are alternatively denoted as *computation workflows* in F#).

The concrete monad is represented as a polymorphic container of value or values of any type (in F# formally typed as `M<'T>`, where 'T is generic type of stored value, hereafter it will be denote as base type). The monad can be based on a general container (lists, closures) but some types of monads requires special structures.

For this container the pair of function must exit:

> function *bind* with type signature: `M<'T> * ('T -> M<'U>) -> M<'U>` and
>
> function *return* with type signature `'T -> M<'T>`

The semantic of function *return* is trivial, it converts the value of base type to monad's container. The semantic of *bind* function is more complicated. In brief, the *bind* function receives a tuple, which contains value stored in monad container (type of this monad is M<'T>) and function. This function converts value of base type to target type (denoted as 'U, type 'U is in simpler monads identical with 'T) and then it wraps results values again to a monad. The bind function return a new monad of type M<'U>.

The detailed semantics of *bind* function is strongly dependent on concrete monads and its container, but all implementations share common feature — they form segments of a production line in which the functions transform values of base types.

For example, the monad based on set container provides semantics of multivalues, i.e values which are elements of power set of base value set. The binary operations on values of base value set (e.g. numbers) can be easily and unambiguously modified (lifted up) to multivalues (analogically for function and operators with other arities):

$$\forall X, Y \in P(V) X \boxplus Y = \{z = x \oplus \mathrm{y} \mid x, y \in V \times \mathrm{V}\} \tag{1}$$

where V is base value set, $\oplus$ is any binary operation on V and $\boxplus$ is uplifted operation in $P(V)$ i.e. on set of multivalues.

The most simple (but nontrivial) multivalues are based on two-elements value sets, e.g. booleans.

The computation expression based on multivalue (= set) monad provides context in which the processing of multivalues is (almost) as natural as processing of values in a normal imperative programming language:

```
result = multivalue {
    let! a [false; true]
    let! b [false; true]
```

```
        return (a && b) || (not a) || (not b)
    }
```

The special forms *let!* are analogy of imperative assignments (or more accurately analogy of symbol binding of functional languages) but with shifted semantics. The symbol *a* (and *b* respectively) are not bounded with set of values (i.e. with collections of values) but with single (multi)value, which simultaneously takes both values true and false i.e. from the point of view of type system the value referred by *a* (and *b* respectively) has type boolean not set of boolean. Therefore the expression after *return* function uses normal (boolean) operators, but it is processed for all possible combinations of states of multivalues (in our example only four combinations exists). Result of each partial processing is wrapped to the set (=i.e. representation of monad) and these monads are unitized into one set (= results monads). This sets is result of computing expression (block inside of braces with header of monad's object) and it is bounded to symbol *result* (in normal functional context, i.e. result does not refer to the multivalue but to normal set).

Note for programmers in F#: the literal `[false; true]` is literal of list not set. But literal of list is briefer (does not require nested computation expression). The value of list is immediately transformed to set (=natural container of multivalue monad).

The syntax of computation expression explicitly displays the function *return*. But the function *bind* (or as in our example the tuple of *bind* function) has not a direct representation. The first parameter (= input monad) is defined by *let!* form, the second (processing function) is implicitly executed as loop over all values of multivalues (i.e. inside the return function) and also as union of partial results wrapped by return (i.e. outside the return function).

The role of bind is better derivable by explicit notation of computation expression, which uses bind function directly:

```
bind [true, false],
    (fun a -> bind [true, false],
        fun b -> return (a && b) || (not a) || (not b)
    )
```

The *bind* function applies its second parameter (function) to all values of is first parameter. Function are represented by lambda expression (anonymous functions) with parameter binded to symbol *a* (outer function) and *b* (inner function).

The *bind* function of this monad is representable by composition of two function: set variant of function *map* (higher-order function which applies a function to all items of set) and set union (the union is in F# defined only as binary operation, i.e. for two sets only, the version with higher arity is constructed by *fold* function):

```
mvBind value f = value |> Set.mapF |> Set.fold Set.union Set.empty
```

Representation of return function is trivial, it constructs single-item set from isolated item (i.e. add item to empty set):

```
mvReturn = Set.add x Set.empty
```

The implementation of monad after definition of its key function is only auxiliary formalism. The function must be linked with header object of computation expression. Unfortunately, this is not attainable by definition of simple OOP class because the *return* function isn't called on monad (i.e. it is not method of monad).

The F# solves this problem via builder class, which formally connects both function and its instance serves as the header object of computation expression (this object is typically singleton).

```
type VariantBuilder() = class
    member this.Bind(x, f) = mvBind x f
    member this.Return(x) = mvReturn x
    member this.Delay(f) = f() //for simple monads always f()
end
let multivalue = VariantBuilder() //creation of header object of comp. expr.
```

The monads still are somewhat esoteric constructs and even with syntactic sugar of computation blocks are used only for customizations targeted to professional coders (i.e. not for practitioners). But for the near future, the monads have the potential to define (and implement) new operational semantics for more specific languages.

4.3 BOO AND SYNTACTIC MACROS

Boo is relatively new language (2003), which uses pythonic syntax for .NET semantics (Boo is native .NET language). The most typical feature of the Boo language is extensive support of metaprogramming, i.e. programming of transformation auxiliary structures of compiler in the phase of compilation.

This approach is not revolutionary, because metaprogramming have been provided by Lisp language from the 1960's (and in more advanced and standardized level from definition of Common Lisp and Scheme dialects of 1980's). But there are important differences:

- Boo isn't homoiconic language (i.e. the language is not implemented on the top of primitive data structures of language). Furthermore, the syntax of language is more similar to mainstream languages than to Lisp dialects.
- Boo is statically typed language with primary OOP support (i.e. macros are based on OOP constructs, no OOP construct on macros, as in Lisp)
- Boo is compiled language (targeted to relatively low level bytecode) not interpreted as Lisp and its dialects. The phase of transformation of program code during compilation is strictly separated from execution phase (running of application)
- the syntactic macros can be implemented in any .NET language (as C#), but they must be consumed only in Boo. However, the implementation in Python is much simpler, because Boo provides crucial syntactic sugar for metaprogramming — quasi-quotation.

The metaprogramming in Boo exploits three basic mechanisms [11]:

1. syntactic macros for implementation of new statement-like constructs
2. implementation of new compilation steps for transformation of existing constructs (including but not limited to metaobject protocol)

3. syntax attributes (analogy of decorators of Pythons, but they are processed in earlier compilation phase and they are based on transformation of syntax trees not on reflection of objects)

Following example of simple DSL in Boo uses only syntactic macros. The syntactic macros have potential to dramatically change universe of language and thus to shield user from original syntax and semantics of Boo language (only *print* statement in example uses original syntax).

```
place Praha:
    latitude: 50|05
    longitude: 14|26

place Berlin:
    latitude: 52|30
    longitude: 13|24

place Wien:
    latitude: 48|12
    longitude: 16|22

print Berlin.latitude
standpoint Praha:
    iterate_other:
        print place, ": distance =",place.distance
```

This DSL is very simplified in order to abridge implementation code but it display some characteristic of modern DSLs:

- **crucial role of declarative notation** of data models (our model is very simple — topographical location (place) with name and geographic coordinates).
- **semantics based on** (possibly nested) **contexts** (evaluation of distance is based on a standpoint)
- **radical simplification of object model** (classes are totally omitted, objects and variables are unified)

The ***declarative data notation*** allows easy initialization of objects with a complex structure. The original approach of programming languages is atomization and linearization of the initialization code or slightly structured data in external (text) files:

```
#code in C language
Place place;
place.name = "Praha"
place.latitude = 50.083 #assignment of atomic value
place.longitude = 14.433

#possible representation in text file (place = one line)
Praha 50.083 14.433
```

Modern languages support more complicated data initializers (e.g. nested constructor in some object languages, in Boo in the form `Place(Latitude("50:05"), Longitude("14:26"))`) but the initialization of really complex objects (e.g. object represented by graph with loops) must be at least partially linearized.

The declarative initializers in Boo can be more compact and closer to a common notation. Unfortunately, some aspects of base syntax must be preserved:

1. surface pythonic syntax with indented blocks and colons after headers (fortunately, this syntax is relatively simple and natural)
2. the place names is to be conform to the identifier pattern of Boo language (i.e. only letters fellowed by letters and digits). The multiword names are representable only with underscores (e.g. *New_York*)
3. the notation must use only standard *Boo* literals (i.e. numbers and strings) or their combinations with standard operators. Therefore the angle in natural sexagecimal notation is represented by two integer literals combined by *bitwise-or* operator (the more appropriate form 15°05 is invalid Boo)

The **context based semantics** is typical for natural human languages and it occurs also in formal languages. The utilization of contexts reduces complexity of language utterances and allows relatively linear representation of hierarchical structures.

The programming languages have used context based semantics by mechanisms of subprograms and OOP methods. More advanced types of contexts are available by some types of polymorphism (implicit casting, OOP polymorphism, generics). These types of context dependency are exploitable only in program which are structured to subprograms or functions. However, the scope of this types of contexts is de facto global, i.e. the context is fixed in a whole program or at least in a subroutine. Local (block scoped) contexts are ordinarily limited to variables (local validity of variables in code blocks) but exceptions exists:

- *try-catch* constructs — limited to context of exceptional situations (i.e. it is not usable for abstract models)
- *using* constructs of C# — limited to allocate and dispose models of external resources
- *context manager* in Python — general solution supported by syntactic sugar (*with* constructs). Currently, context managers are used for resource manager (like *using* constructs) or low-level synchronization (i.e. only for infrastructural purposes).

In sphere of domain specific languages, the using of contexts is much broader. The contexts are not limited to infrastructural and low level constructs but they are related to a targeted model. In our simple DSL the block *standpoint* create natural context which is typical for navigation application: the implicit point from which distances and azimuths are measured (standpoint, point of observer).

The target point is defined by nested context of place iterator (*iter_other* construct). This context is more explicit (the target point is identified by identifier *place*) and it is also optional (the target is alternatively identifiable by direct object specification).

Implementation of new language constructs is relatively straightforward in Boo because the language provides high-level constructs for metaprogramming — syntactic macros and *quasi-quotations*:

```
macro place:
    macro latitude:
        place["lat"]=latitude.Body.ToString()
    macro longitude:
         place["lng"]=longitude.Body.ToString()
    symbol = place.Arguments[0]
    name = Expression.Lift(symbol.ToString())
    lat = place["lat"] as string
    lng = place["lng"] as string
    return [|
        block:
            places[$name] = Place($name, $lat, $lng)
            $symbol = places[$name]
    |].Body


macro standpoint:
    name = Expression.Lift(standpoint.Arguments[0].ToString())
    return [|
        ContextManager.default.EnterContext(places[$name])
        $(standpoint.Body)
        ContextManager.default.LeaveContext()
    |]

macro iterate_other:
    return [|
        for place as Place in places.Values:
            if place != ContextManager.default.Place:
                $(iterate_other.Body as Block)
    |]
```

The macro *place* converts structured block of data initializer to assignment statement for new created object. The structured block contains two parameterizable parts: arguments (in header between macro name and colon) and body (indented block after header or in case of single-line blocks code after colon). The processing of body of *place* structured block is provided by two nested macros: *latitude* and *longitude*. These macros store their bodies (angle values in form of simple expressions) into *standpoint* macro object (macro is object with dictionary-like interface) but they do not produce any new code in form of abstract syntax trees.

The main macro utilizes these stored values in its own transformation. This transformation replaces the original AST (created from macro call) by new syntax tree which is obtained from parsing of quasi-quotation. The quasi-quotation is delimited by "[|" and "|] parentheses and forms argument of *return* statement. Within quasi-quotation the symbols, preceded by dollar sigil, are substituted by values of particular variables (the variables exists only in phase of macro substitution). Note especially, difference between symbol and string literals in AST (symbol containing place name must be transformed [lifted] to string literal in context of dictionary indexing). The whole process is depicted in Figure 3.

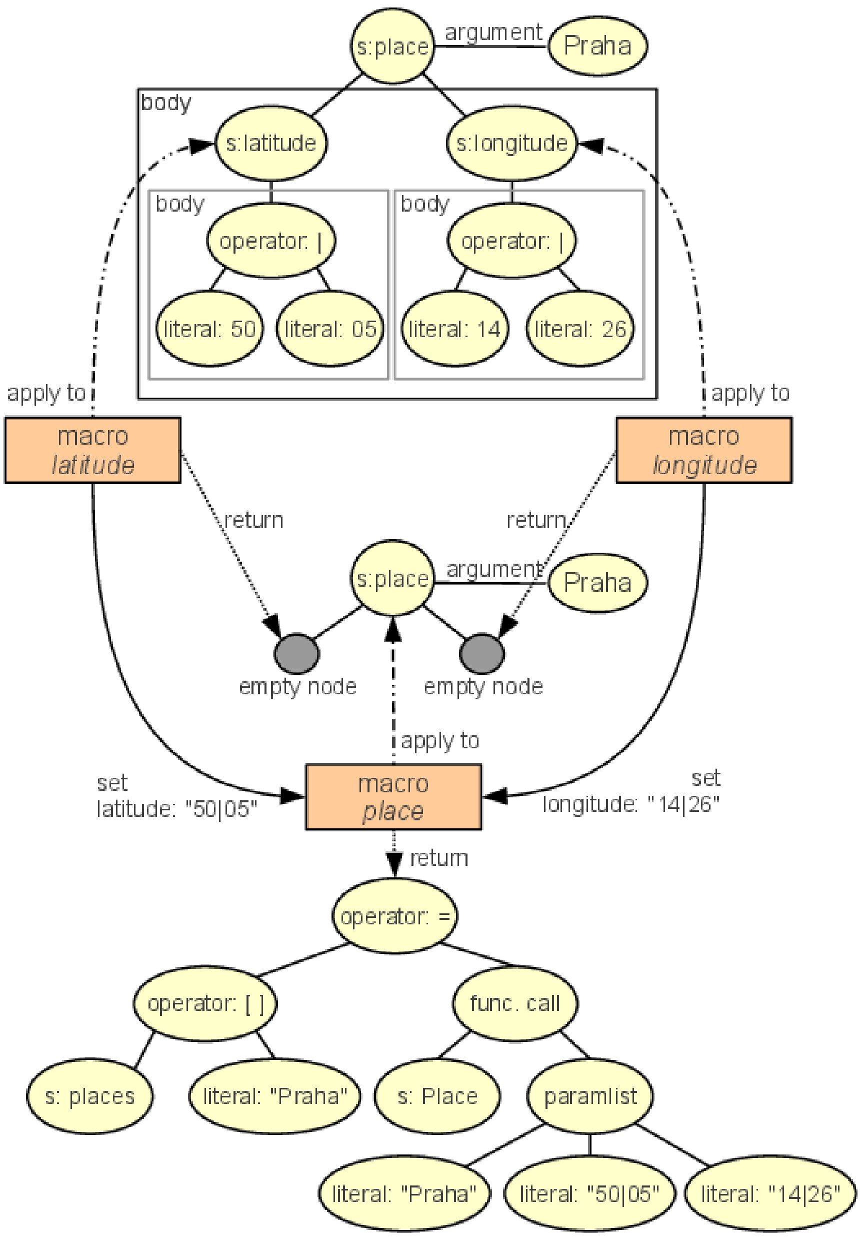

Fig. 3. Transformation of macro place (example for argument Praha).

The second macro (*standpoint*) uses simplified context manager in newly generated code for storing of contextual data on the stack (context manager is defined in supporting OOP library). The third macro maps macro *iterate_other* to loop over all places (excluding current standpoint).

The implementations of the macros aren't robust. The macros do not check their arguments and bodies and the syntax of the new DSL has some awkward limitations (e.g. only one statement is supported in *iterate_other* body). The increasing of robustness requires additional (wearisome) code.

Boo is relatively marginal language but some principles of metaprogramming are applicable on all platforms which provide *compiler as service* (in .NET there exists two projects *CSharpRepl* in *Mono* [3] and *Roslyn* of *Microsoft*, [5]). The definition of new language is complicated without high-level API and syntactic sugar of Boo but not impossible. Furthermore, the support for metaprogramming can be offered in the form of third part libraries and compiler extensions. The compilers cease to be black-boxes which product only byte or machine codes in one fixed way. The modifications or at least tunings of compilers make possible universal support of metaprogramming and everyday design of specific end-user languages.

5 CONCLUSIONS

The contemporary domain-specific languages are relatively simple and restricted to models, which are close to standard programming. Even the simplified universal formal models are not directly representable by programming languages (e.g. economic or physical models) and the practitioners are forced to transform unnaturally the model to functional or imperative algorithms (with help of programmers).

There are several reasons of this condition:

- the active programmers (including some practitioners) are affected by programming languages which use. This natural feedback constrains their thinking to paradigm or constructs of these languages
- design and implementation of new domain specific language is arduous (time consuming, expansive), especially from scratch

The implementation of new domain specific language by modification and accommodation of existing programming language solves at least partially those problems because the new language can share some syntactic and semantics properties with its parent, i.e.

- *the design of new language may reuse some syntactic and semantics structures of parent language* (in consistent way), see Python example in section 4.1.
- *some constructs and entities of parent language* (e.g. statements or classes) *may be directly mapped to new constructs and entities of DSL* (forming some kind of homomorphism)
- *the new language may form only isolated code island in parent language* (i.e only the part of a model is represented by ad hoc DSL). Both parts are able communicate each other by using of homomorphism of data structures. For example applications of monads (section 4.2) form only small islands in code of parents language and there exists simple data homomorphism.
- *the new language may support embedded code in parent language* (again they communicate using data homomorphism). See Boo example in section 4.3 (new DSL uses embedded *print* statement and real number expressions)
- *the supported code* (e.g. macros, decorators, class definition, etc) *may be implemented in context of parent language* (i.e. using its compiler, libraries).
- *the language may be extended by end-users* (by code in parent language, but there is possibility of self-governed extensions)

Unfortunately, the final goal i.e. natural thinking in programming language requires prerequisites which aren’t easily achievable in any contemporary platform.

1) high level support for language design (i.e. not only for implementation)

The new DSL is not only collection of new constructs and extension and must be designed as a whole. The documentation of new language (including grammar) should be automatically inferable from supporting (meta)code or vice versa (metacode from grammar). Contemporary metaprogramming tools do not offer this functionality at all. On other hand the automatic generation of standalone parsers from grammar rules is supported by plenty of tools (*ANTLR, yacc*, etc).

2) metaprogramming support in main stream languages in the form of well tested and documented tools

The metaprogramming support is currently available only in peripheral languages and/or in the form of peripheral constructs. The metaprogramming in Boo is poorly documented and metaobject protocol of Python or F# monads are not included in basic courses of university curricula (and in extended courses these constructs are often mentioned only in pure theoretical context)

3) tool for designing and testing dependability of new DSL

The every really usable DSL must be robust and consistent i.e. any construct of new language must be either valid or product well defined error (with semantic of new domain specific not parent language). The contemporary metaprogramming tools support only low level checking constructs (if statements and asserts) and classical testing tools (e.g. unit testing). The most problematic aspect of metaprogramming — two-phase execution of code must be reflected in design of testing tools (analogy of static and dynamic checking of code of universal languages).

This situation has resulted from initial stage of metaprogramming approach (at least in area of non-Lisp languages) and in near future we can expect significant progress in this area. The new domain specific languages based on this progress may make thinking in programming language a common (and even basic) tool of scientific modeling and thinking.

REFERENCES

[1] DAHL, O.-J., NYGGARD, K. *Simula — A language for programmingt and decription of discrete event systems. Introduction and user's manual.* 1967. URL http://www.edelweb.fr/Simula/slp-0.pdf

[2] DIJKSTRA, E.W. *How do we tell truths that might hurt?* 1975. URL http://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD498.html

[3] ICAZA, M. *Mono's C# Compiler as a Service on Windows.* 2010. URL http://tirania.org/blog/archive/2010/Apr-27.html

[4] KICZALES, G., ASHLEY, J.M., RODRIGUEZ, L., VAHDAT, A., BOBROW, D.G. *Metaobject protocols: Why we want them and what else they can do.* Object-Oriented Programming: The CLOS Perspective, pp. 101-118, 1993.

[5] KUNK, J. *10 Questions, 10 Answers on Roslyn.* Visual Studio magazine, 2012. URL http://visualstudiomagazine.com/articles/2012/03/20/10-questions-10-answers-on-roslyn.aspx

[6] MCCREA, A. *Metaprogramming JavaScript*. URL http://s3.amazonaws.com/edgecase/metaprogramming_js.pdf

[7] O'SULLIVAN, B., GOERZEN, J., STEWART, D. *Real World Haskell: Code You Can Believe*. In: O'Reilly Media, Incorporated, 2008.

[8] ODERSKY, M., SPOON, L., VENNERS, B. *Programming in Scala*. Artima, 2008. 756 p. ISBN: 978-0981-531-601.

[9] PERROTTA, P. *Metaprogramming Ruby: Program Like the Ruby Pros*. Pragmatic Bookshelf, 2010. 296 p. ISBN: 978-1-93435-647-0.

[10] PETRICEK, T., SYME, D. *Syntax Matters: Writing abstract computations in F#*. In pre-proceedings of TFP, 2012. Draft paper.

[11] RAHIEN, A. *DSLs in Boo: Domain Specific Languages in .NET*. Manning Publications, 2010. 352 p. ISBN: 978-1933-988-603.

[12] PICKERING, R. *Beyond Foundations of F# - Workflows*. InfoQ, 2008. URL http://www.infoq.com/articles/pickering-fsharp-workflow

[13] PECINOVSKÝ, R., PAVLIČKOVÁ, J., PAVLÍČEK, L. *Let's Modify the Objects-First Approach into Design-Patterns-First*. In: ITiCSE, 2006. URL http://edu.pecinovsky.cz/papers/2006_ITiCSE_Design_Patterns_First.pdf

[14] SHEARD, T., JONES, S.P. *Template meta-programming for Haskell*. In: ACM SIGPLAN workshop on Haskell. pp. 1—16, 2002. DOI: http://doi.acm.org/10.1145/581690.581691

[15] SUMMERFIELD, M. *Advanced Python 3 Programming Techniques*. Pearson Technology Group, 2009. 97 p. ISBN: 978-0-321-63772-7.

[16] TRATT, L. *Domain specific language implementation via compile-time meta-programming*. ACM Trans. Program. Lang. Syst., pp. 31:1—31:40, 2008. DOI: http://doi.acm.org/10.1145/1391956.1391958

[17] VILLAR, J.I. *Python as a hardware description language: A case study*. In: Programmable Logic, pp. 117-122, 2011. DOI: http://dx.doi.org/10.1109/SPL.2011.5782635

[18] WIRTH, N. *Algorithms + data structures = Programs*. Prentice-Hall, 1976.